A Generic Framework for Symbolic Execution:Theory and Applications

Andrei Arusoaie

A Generic Framework for Symbolic Execution: Theory and Applications

by

Andrei Arusoaie

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy in Computer Science

Faculty of Computer Science
Alexandru Ioan Cuza, University of Iaşi

2014
Iaşi, Romania

Doctoral committee:

Professor Dr. Dorel Lucanu, *Supervisor, UAIC*

Associate Prof. Dr. Adrian Iftene, *Chair, UAIC*
Associate Prof. Dr. Marius Minea, *Politehnica University of Timişoara*
Researcher Dr. Vlad Rusu, *INRIA Lille-Nord Europe*
Professor Dr. Gheorghe Ştefănescu, *University of Bucharest*

A Generic Framework for Symbolic Execution: Theory and Applications

Andrei Arusoaie

Abstract

The modern world is shifting from the traditional workmanship to a more automated work environment, where software systems are increasingly used for automating, controlling and monitoring human activities. In many cases, software systems appear in critical places which may immediately affect our lives or the environment. Therefore, the software that runs on such systems has to be safe. This requirement has led to the development of various techniques to ensure software safety.

In this dissertation we present a language-independent framework for *symbolic execution*, which is a particular technique for testing, debugging, and verifying programs. The main feature of this framework is that it is parametric in the formal definition of a programming language. We formally define programming languages and symbolic execution, and then we prove that the feasible symbolic executions of a program and the concrete executions of the same program mutually simulate each other. This relationship between concrete and symbolic executions allow us to perform analyses on symbolic programs, and to transfer the results of those analyses to concrete instances of the symbolic programs in question. We use our symbolic execution framework to perform program verification using Hoare Logic and Reachability Logic. For the latter, we propose an alternative proof system, and we show that under reasonable conditions, a certain strategy executing our proof system is sound and weakly complete.

A prototype implementation of our symbolic execution framework has been developed in $\mathbb{K}$. We illustrate it on the symbolic execution, model checking, and deductive verification of nontrivial programs.

Contents

"*Every beginning is a consequence*
– every beginning ends something."
Paul Valery

Chapter 1

Introduction

This dissertation proposes a language independent framework for symbolic execution, based on formal definitions of programming languages and used for program analysis and verification.

1.1 Motivation

Programmable electronic devices have made our lives more efficient, pleasant, and convenient. Most of them are designed for personal usage (e.g. laptops, smartphones, tablets, etc.) and significantly improve communications and the information flow. Other devices have been adapted for monitoring and controlling industry automated systems, avionic flight management units, nuclear reactor control systems, spaceflight control, and so on. Many of them can be found in places which immediately affect our daily activities, like automotive braking systems, robotic surgery machines, or traffic control devices. No matter what the destination of an electronic device is, the most relevant component is the software that runs on it. For such critical systems, software has to be *safe* in order to avoid harmful situations. The growing demand for this type of systems has led to the development of various techniques to ensure software safety. A particular technique is *symbolic execution*, which has been used for testing, debugging, and verifying programs written in various programming languages. The concept mainly consists in sending *symbolic values* as inputs for programs instead of concrete ones, and the execution is done by

manipulating expressions involving symbolic values. A symbolic program execution typically memorises symbolic values of program variables and a *path condition*, which accumulates constraints on the symbolic values on the path leading to the current instruction. When the next instruction to be executed is a conditional statement, which depends on symbolic values, the execution is separated into distinct branches. The path condition is then updated to distinguish between the different branches. All possible executions can be merged together into a *symbolic execution tree*, where each node is associated with a *symbolic state* and each edge with a transition between these states. Symbolic states typically include the statements to be executed, the program counter, the set of variables and their values, and the current path condition.

The main advantage of symbolic execution is that one can reason about multiple concrete executions of a program at once. On the other hand, its main disadvantage is the state space explosion caused by decision statements and loops. Recently, the technique has found renewed interest in the research community due to the progress in decision procedures, new algorithmic developments, and availability of powerful computers.

Currently, a large number of tools use symbolic execution for different purposes: automated test case generation, program verification, static detection of runtime errors, predictive testing, etc. Despite the fact that symbolic execution is a language independent technique, the existing symbolic execution engines target specific programming languages. As a consequence, it is quite hard to extend or reuse them for a new language. Often, a new implementation is preferred rather than using an existing one. An alternative is to translate programs from one language to another and then run symbolically the transformed programs. However, this solution may have implications on different analyses because the obtained results depend on the translation. Following this approach, some tools implement symbolic execution for assembly languages. Then, compilers are used to translate programs into assembly code. Symbolic execution of a given program is actually done by executing the corresponding assembly program, not the initial one. Thus, one has to be sure that the semantics of the original program is preserved during compilation, otherwise symbolic execution may not explore the desired execution tree.

A non-negligible aspect, that the majority of the symbolic execution

tools are less concerned about, are formal definitions of programming languages. Current implementations are mainly based on existing compilers or interpreters. Due to the fact that standard manuals for programming languages usually contain informal descriptions, compiler developers may have a different understanding of the semantics of a language. Therefore, different compilers can exhibit different behaviours of the same program. This has a negative impact on symbolic execution too, since any compiler can possibly explore different execution paths of the same program. As a consequence, any results obtained by analysing programs using symbolic execution are not transferrable across platforms that use a different compiler. In contrast, formal semantics provides an exact (mathematical) meaning for each language construct, and symbolic execution based on formal semantics captures all possible behaviours of a program.

The aim of this dissertation is to create a language independent formal framework and a tool for symbolic execution which is based on the formal definition of a programming language. There are several advantages of this framework over the existing approaches. First, a generic framework for symbolic execution can be reused for any programming language, without additional knowledge about its implementation. Second, the framework is based on the trusted formal semantics of a language, and thus, misinterpretations of language constructs is avoided. Finally, in a formally defined framework for symbolic execution, the relationship between concrete and symbolic execution can be also formalised. This relationship essentially states if symbolic execution can be used as a sound program analysis technique. Additionally, a practical impact of this framework is modularity, since the symbolic execution engine is a standalone tool, which can be easily integrated by other tools.

1.2 Contributions

Here is a summary of the main contributions of the research presented in this dissertation:

1. The main contribution of this dissertation is a formal, language-independent theory and tool for symbolic execution, based on a language's operational semantics defined by term-rewriting.

2. On the theoretical side, we define symbolic execution as the application of rewrite rules in the semantics by *symbolic rewriting*. We prove that the symbolic execution thus defined has the following properties, which ensure that it is related to concrete program execution in a natural way:

   Coverage: for every concrete execution there is a feasible symbolic execution on the same program path, and

   Precision: for every feasible symbolic execution there is a concrete execution on the same program path,

   where a symbolic execution is feasible if the path conditions along it are satisfiable. In terms of simulations, we say that the feasible symbolic executions and the concrete executions of any given program mutually simulate each other.

3. On the practical side, we present a prototype implementation of our approach in $\mathbb{K}$ [97], a framework dedicated to defining formal operational semantics of languages. Since the current version of $\mathbb{K}$ is based on standard rewriting, we show how symbolic rewriting can be achieved by applying certain modified rewrite rules (obtained by automatically transforming the original ones) in the standard manner over a symbolic domain.

4. We show that our symbolic execution framework can be used for verifying programs using Hoare-like Logic. The approach consists in translating Hoare triples into particular Reachability Logic [100] formulas, which can be automatically proved using symbolic execution and an SMT solver.

5. We present an automatic, language-independent program verification approach and prototype tool based on symbolic execution. The program-specification formalism we consider is Reachability Logic, a sound and relatively complete deduction system, which offers a lot of freedom (but very few guidelines) for constructing proofs. Hence, we propose an alternative proof system, in which symbolic execution becomes a rule for systematic proof construction. We show that, under reasonable conditions on the semantics of programming

languages and of the Reachability-Logic formulas, a certain strategy executing our proof system is sound and weakly complete.

1.3 Related work

In this section we present some of the approaches/tools that use/implement symbolic execution for various applications. We identify the key ingredients of symbolic execution (exhibited by previous research on this topic) which are incorporated into our framework, and we emphasise the need for a language independent framework, which should be based on the formal semantics of programming languages. Note that we focus more on the symbolic execution basic features exhibited by various tools and less on applications.

In 1976, James C. King presents one of the first tools for automated test generation (EFFIGY) based on symbolic execution. In his paper [62], King intuitively describes symbolic execution as follows:

"*Instead of supplying the normal inputs to a program (e.g. numbers) one supplies symbols representing arbitrary values*".

The key concepts of symbolic execution introduced in [62] are: *symbolic values*, *path conditions*, *symbolic states* and *symbolic execution trees*. Briefly, symbolic values are nothing else than variables of a given type (e.g. integers), path conditions are logical formulas involving symbolic values, and symbolic execution trees are program executions merged together in a tree, where each branch is induced by a conditional statement and nodes are *symbolic states*. Let us consider that α is an integer symbolic value. The value of α can be any integer, but it is not known which one. For this reason, α can be considered and it will be used as an integer constant. When such values are provided as inputs to programs, the execution semantics of the program is changed into *symbolic execution semantics*. The symbolic execution of regular assignments is quite natural: the expression on the right hand side is evaluated and the result is assigned to the variable. For instance, if `a` and `b` are two variables, where `b =` α and `a = b + 1`, then the evaluation of the second assignment will attach to `a` the sym-

bolic value α `+ 1`. Thus, program variables are assigned with *symbolic expressions*, i.e. expressions involving symbolic values.

A "state" of a symbolic program execution contains the current statement, symbolic values of program variables, the program counter, and the current path condition. The states are connected through edges which represent the transition from one state to another. When the control flow of a program depends on symbolic values, say an IF-THEN-ELSE statement, then there are more possible choices, one for each branch of the statement. For instance, let pc be the current path condition an q be the condition of IF-THEN-ELSE. If q is a symbolic expression then we are in one of the following cases (we use $\rightarrow$ for the logical implication):

- either $pc \rightarrow q$ holds and then the execution continues on the THEN branch;
- either $pc \rightarrow \neg q$ holds and the execution follows on the ELSE branch;
- neither expression $pc \rightarrow q$ nor expression $pc \rightarrow \neg q$ holds; in this case there is at least one set of inputs which satisfies the pc and would take the THEN branch, **and** there is at least one set of inputs which satisfies the pc and would take the ELSE branch.

In the last case the execution *forks* into two "parallel" executions, and thus, from the current state there are two transitions in the symbolic execution tree. A path condition has the property that it determines a unique control flow path from the program. Initially, the path condition has the initial value set to "true", and it is constructed during the execution by collecting branch conditions (q): $pc = pc \wedge q$ and $pc = pc \wedge \neg q$. Note that a path condition constructed as above is always a satisfiable formula, since for every execution there are some input values which led the execution to that specific state. Thus, there are two important properties that path conditions can have:

- pc can never become identically "false" because there does exist a concrete input to the program which, when executed normally, will trace the same path.
- If pc_1 and pc_2 are two path conditions then $\neg(pc_1 \wedge pc_2)$ holds. If pc_1 and pc_2 have a common part then, there exists, by construction,

a state which caused the two paths to be split by adding q and $\neg q$ to the common part of the two paths.

Another property of symbolic execution exhibited in [62] is that if all symbolic values are instantiated by concrete values then the result is exactly the same as the one obtained when executing the program with the same concrete values. This property (also called *commutativity* in [62]) states that symbolic executions capture precisely concrete executions, and this is why symbolic execution is of interest also in program verification. The author presents a methodology for proving programs correct using the Floyd's method [42]. Three new ancillary statements (ASSUME, ASSERT, and PROVE) are introduced in the language in order to help with the proving process as follows: the user introduces assertions (statements) of the form ASSERT(b), where b is the formula expected to be valid at (carefully chosen) specific points in the program, such that a fixed set of paths, which start and end with an ASSERT, is obtained. Now, one must show that using *any* set of variable values which satisfy the assertion at the beginning of the path, the execution resulting along the path must satisfy the predicate at the end. This can be easily achieved by symbolic execution as follows:

1. change the ASSERT at the beginning of a path with an ASSUME, and the ASSERT at the end with a PROVE statement
2. initialize *pc* to *true* and all program variables with symbolic values
3. perform symbolic execution as usual;
4. in the end, execute the PROVE statement and if it displays *true* then the program is verified successfully.

Using this technique, the author claims that for any program, which has a finite symbolic execution tree and the correctness criteria is made explicit with assertions, the exhaustive symbolic execution and the proof of correctness (done following the steps above) are exactly the same process.

EFFIGY, the system described in [62], is among one of the first tools which implemented symbolic execution (there were also other systems developed in parallel: SELECT [19], DISSECT [56], and Clarke's system [28]). It is able to execute programs of a simple language whose

statements are assignments, conditional statements, GOTOs, and statements for dealing with input/output operations. The only supported symbolic values are integers. From the beginning, EFFIGY was designed as an interactive tool which allows step-by-step execution, tracing and state manipulation (load/save/edit). To handle state space explosion, EFFIGY allows users to either limit (bound) the state exploration giving the maximum number of statements to be executed or, in the interactive mode, to decide when to stop the execution. Moreover they have the freedom to explore the execution tree by navigating in any direction (up/down, left/right). Due to its practical limitations, EFFIGY has been used more like an experimental tool. However, considerable insight into the general notion of symbolic execution has been gained during its development.

Symbolic execution presented in [62] served as the main specification for the development of the symbolic execution framework presented in this dissertation. As in [62], our approach deals with symbolic data, computing path conditions for each program state, and generating the symbolic execution tree using the same principles. On the other hand, we generalize symbolic execution in a few directions. First, we do not restrict de data domain only to integers, users having the freedom to decide the symbolic values domain. Second, we implement a language independent symbolic execution framework, so we do not stick to a particular language. Finally, we formally define symbolic execution and prove the Coverage and Precision properties, which formalize the *commutativity* property intuitively presented in [62]. Based on these properties, we have shown in Chapter 4 how this framework can be used for program verification. Particularly, we present in Section 4.1 a similar (but more generic) approach to the one in [62]. In terms of implementation, our tool for symbolic execution covers all the basic features of EFFIGY (visualisation of symbolic execution paths, displaying the whole execution tree, and interactive exploration of the execution tree).

A more complex symbolic execution tool is Symbolic (Java) PathFinder [81] (or SPF). SPF is an extension of Java PathFinder [82] (JPF), whose development started in 1991 as a front end for the SPIN model-checker [55]. Initially, JPF was designed to model-check Java programs. The latest de-

velopments transformed it in more than a model checker; it became an extensible JVM which can be used as a workbench for efficient implementations of analysis and verification tools. Unlike the experimental EFFIGY tool, JPF is more practical, being successfully used for test case generation [61], invariant detection (using iterative invariant strengthening) [83], predicate abstraction [33], model-checking for Java byte code [84], etc.

SPF has been successfully used to generate tests for a component of a NASA flight software system, achieving full test coverage, for a special coverage metric required by the developer. The basic idea behind SPF is to use the JPF's model-checker to generate the state space of a Java (bytecode) program and to collect the path condition for every state. This is the equivalent of generating the symbolic execution tree for the program in question. For a state s, the symbolic execution tree contains a satisfiable path condition. The satisfiability ensures the existence of a *model* (i.e. a set of concrete values for the variables occurring in the path condition for which the path condition holds). SPF obtains such a model using either an internal constraint solver or an external SMT solver. The model itself is actually a test, in the sense that its concrete execution follows a path in the control flow graph until it reaches state s. To avoid state space explosion, JPF performs bounded model-checking, up to a specific depth.

Being developed for a real life language, SPF revealed new practical challenges raised by symbolic execution, like support for complex data structures (lists, trees, arrays), inheritance, polymorphism, the need for powerful SAT, SMT or constraint solvers, etc. During the development of our language independent framework, we have considered all these challenges and we reused most the solutions proposed for SPF. However, there are some features which cannot be covered by our framework by default, since they purely depend on the language or the implementation platform. For instance, SPF permits symbolic execution to be started at any point in the program and at any time during concrete execution. This allows symbolic execution for specific parts of the program. Moreover, parts of the program can be annotated with pre-conditions that help improving the precision of the unit-level symbolic analysis (e.g. for avoiding generation of tests that violate the precondition). In our framework such features are not present by default (since we do not have a generic way to instrument when to switch from symbolic to concrete execution) but we offer means

to implement them. On the other hand, SPF is hard to extend or to reuse for implementing symbolic execution for a different language. From the theoretical point of view, SPF does not concern about defining symbolic execution or relating it with the concrete execution.

There are some modern challenges of symbolic execution which are related mostly to library classes and native code. Such code needs an explicit modelling to make symbolic execution possible. One approach, which is meant to trick the explicit modelling by avoiding it, is combining concrete and symbolic execution, also known as *concolic* execution. This type of hybrid analysis, which performs both concrete and symbolic execution has been implemented by several tools in order to get dynamic test generation. The main point of concolic execution is to perform concrete and symbolic execution together which outputs inputs to cover the "new" behaviour. For instance, consider that we first run a program p with concrete values for the input variables x and y. The execution follows naturally until, at some point, a conditional statement has to be executed. Here, the statement's condition, say $x \leq y$, can be evaluated as usual, since we know the concrete values of x and y; thus, the concrete execution may continue on one branch or another. In parallel, we also collect the condition which has to be satisfied by the variables such that the execution will follow the other path, i.e. $\neg(x \leq y)$. For the next run, we choose concrete values for x and y such that $\neg(x \leq y)$, and this will allow us to explore a different program path. This a *directed* symbolic execution, the new program paths and corresponding path conditions being discovered using concrete executions. The technique is also known as *dynamic symbolic execution* and it is implemented by many tools for test case generation, e.g. DART [45], CUTE [107], jCUTE [106], EXE [24], PEX [38] SAGE [46], CREST [22]. Among these, only PEX and SAGE are used at large scale. PEX is a test case generation tool implemented for Microsoft's .NET platform and is distributed with Visual Studio. It performs a systematic program analysis (using dynamic symbolic execution) to determine test inputs for parameterised Unit Tests. PEX learns the program behaviour by monitoring execution traces. It uses a constraint solver to produce new test inputs which exercise different program behaviour. The result is an automatically generated small test suite which often achieves high code coverage.

A similar tool, also developed at Microsoft, is SAGE which uses whitebox fuzz testing that generates tests for x86 assembly code. The tool executes the program under test with a set of concrete valid inputs and records the explored path. Then, it evaluates symbolically the recorded path and gathers for constraints on inputs capturing how the program use these. The collected constraints are negated one by one in order to obtain new input values using an SMT solver. SAGE found many security bugs in Windows applications.

Other approaches [103, 20] use dynamic symbolic execution to detect errors in programs written in interpreted languages, e.g. Python. The are many challenges when performing symbolic execution on interpreted languages. For instance, statements in programs can wrap complex operations, so expressing path conditions as precise as possible requires a complex theory. Additionally, due to the fact that most of these languages are dynamically typed, symbolic execution has to handle type information also. Some variations of concolic execution consist in performing first symbolic execution, and then concrete execution. The reason is that the symbolic execution will first detect imprecisely inputs which produce errors and the concrete execution will filter the initial results.

Our framework supports both concrete and symbolic execution. For concrete inputs the tool acts as expected in a concrete run of the program. Thus, one can develop different variations of concolic execution.

Another body of related work is symbolic execution in term-rewriting systems. The technique called *narrowing*, initially used for solving equation systems in abstract datatypes, has been extended for solving reachability problems in term-rewriting systems and has successfully been applied to the analysis of security protocols [78]. Such analyses rely on powerful unification-modulo-theories algorithms [37], which work well for security protocols since there are unification algorithms modulo the theories involved there (exclusive-or, ...). This is not always the case for programming languages with arbitrarily complex datatypes.

All the aforementioned approaches are based on symbolic execution engines developed for specific programming languages. However, all of them follow the same principles of symbolic execution stated in [62]. A question which naturally arises in here is: why not having a tool which is able to run symbolically programs of any programming language? An approach which

might answer to this question is KLEE [23] (which generalises EXE [24]), a symbolic virtual machine build on top of the LLVM [65] compiler infrastructure. Since there is a symbolic virtual machine for LLVM and compilers from different languages (C, C++, Objective-C, Ruby, Python, Haskell, Java, D, PHP, Pure, Lua) to this virtual machine then, as a consequence, symbolic execution is available for any of these languages.

In terms of usability, KLEE proved its usefulness, especially in tools for automated test case generation and bug detection. It has been used to detect errors in GNU COREUTILS (version 6.10). The results were quite impressive, the tool achieving the highest code coverage (84.5% of the total number of lines of C code), and discovered ten unique bugs. One of them was introduced back in 1992, and has not been discovered for 16 years, even if COREUTILS was analysed by lots of tools, before KLEE.

However, there is a critical issue which is not addressed by KLEE. What is supposed to be its main advantage, namely the symbolic virtual machine built on top the LLVM compiler, is also one of its main disadvantages. When performing symbolic execution for a particular language, say $\mathcal{L}$, one expects the execution to happen as specified in the semantics of $\mathcal{L}$. However, in KLEE, $\mathcal{L}$ programs are compiled into LLVM, and then, their execution depends on the semantics of LLVM. This could be a problem, since the semantics of $\mathcal{L}$ and LLVM may differ for some language constructs, and the KLEE does not treat such cases. A very simple example is the semantics of the + operator in C, which does not specify an evaluation order for the operands of +. If those operands are functions with side effects, then the evaluation from left to right might produce a different result than the evaluation from right to left. The LLVM compiler for C assumes a default evaluation order for the arguments of +, and thus, when performing symbolic execution the tool will explore a single path instead of two paths (coresponding to evaluation from left to right and evaluation from right to left). When performing some analysis based on symbolic execution of a program, then the results of the analysis might not be valid anymore when using a different compiler. Therefore, there is no guarantee that the translation from $\mathcal{L}$ to LLVM preserves the semantics of $\mathcal{L}$'s constructs. This approach makes symbolic execution compiler dependent. On the other hand, the symbolic execution framework proposed in this dissertation is parametric in the formal semantics of a programming

language. We perform symbolic execution using *directly* the semantics of the language, no translations being involved in the process.

Recently, rewriting modulo SMT was proposed as a new (language independent) technique which combines the power of SMT solving, rewriting modulo SMT, and model checking, for analysing infinite-state open systems [89, 90]. An open system is modelled as a triple (Σ, E, R), where (Σ, E) is an equational theory describing the system states and R is a set of rewrite rules describing the system's transitions. The state of an open system must include the state changes due to the environment. These changes are captured by new fresh symbolic variables introduced in the right hand side of the rewrite rules. Thus, the system states are represented not as concrete states, but as symbolic ones, i.e. terms with variables (ranging in the domains handled by the SMT solver) which are constrained by an SMT-solvable formula. Rewriting modulo SMT can symbolically rewrite such states (which may describe possibly an infinite number of concrete states). The approach from [90] is very similar with the one presented in this dissertation, since both of them deal with symbolic states, symbolic rewriting, and use SMT solvers to decide the satisfiability of path conditions. Moreover, the soundness and completeness results from [90] are very similar with our coverage and precision properties. On the other hand, the approach allows symbolic execution modulo axioms, which is not supported by our framework. However, this feature comes with hard conditions that the model used for interpretation has to meet.

A major application of symbolic execution that we are interested in this dissertation is program verification. Almost all of the tools performing verification are specialised for a particular verification target. For instance, BitBlaze [105] is a powerful binary analysis platform designated to detect malicious code and/or prevent it. This platform uses symbolic execution directly on binary code for verifying if particular program paths have security vulnerabilities (e.g. buffer overflows).

S2E [25] is a platform for analysing properties and behaviour of software, which is based on two ideas: *selective symbolic execution* and relaxed execution consistency models. Selective symbolic execution is a way to automatically minimise the amount of code to be executed symbolically, given a target analysis. The tool [26] has been used to verify different

properties of systems, e.g. check if memory safety holds along all critical paths, verify if the code that handles license keys in proprietary program is safe when used with different keys, etc. S2E allows users to specify what code to explore paths for, and treats the other code as an "environment" which runs concretely. It includes a set of heuristics to deal with the transitions between concrete and symbolic executions. S2E also supports two interesting features of symbolic execution that we have not addressed in this dissertation: use a cluster for speeding up symbolic execution and state/path merging [64]. CHEF [21] is a platform for obtaining symbolic execution engines for interpreted languages which relies on S2E. CHEF reuses the interpreter as an executable language specification, which reduces the effort of creating a symbolic execution engine. The authors of CHEF have also implemented engines for Python and Lua, and then used them to generate tests for popular code libraries. An interesting feature of this tool is that it handles very well native implementations for functions that are present in interpreted languages. For instance, in Python strings library there are lots of functions (e.g. `find`) which are natively implemented by the interpreter. When performing symbolic execution, CHEF also explores the internal code for these functions, and thus, it increases the tests coverage. In contrast, the symbolic execution framework that we propose in this dissertation handles only those language constructs that have formal semantics. Therefore, for native functions, one has to provide the code and the semantics of the language that code has been written in.

The systems code must obey many rules, e.g. assertions must succeed, allocated memory must be freed, opened files must be closed, etc. WOODPECKER [31] is a tool which verifies such rules in C programs. It explores the symbolic execution tree in order to find bugs, but in an optimised manner: instead of exploring the whole tree, the tool *prunes* it in a *sound* manner depending on the rules to be verified. This is a form of path slicing. Unlike S2E, WOODPECKER is only designed to check rules and it can probably make use of some features of S2E, e.g. state merging or speeding up the execution using a cluster. On the other hand, S2E may use the WOODPECKER's capabilities to guide symbolic execution or to prune execution paths.

Some of the existent approaches use a combination of symbolic execution and static analysis to check whether a particular line of code is

reached [72, 112]. Symbolic execution is sometimes used to only validate feasible paths discovered a priori using static analysis of the program's data and control dependencies [86]. This is the opposite of WOODPECKER's approach, which uses first symbolic execution and then performs static analysis. Other known tools for program verification are [29], [60], and [87]. Several techniques are implemented to improve the performance of these tools, such as *bounded verification* [27] and *pruning* the execution tree by eliminating redundant paths [31]. The major advantage of these tools is that they perform very well, being able to verify substantial pieces of C or assembly code, which are parts of actual safety-critical systems. On the other hand, these verifiers hardcode the logic they use for reasoning, and verify only specific programs (e.g. written using subsets of C) for specific properties (e.g. allocated memory is eventually freed). In contrast, our goal is to create a generic platform for symbolic execution where such applications can be developed with less effort and in a reasonable amount of time. In this dissertation we present a general approach for program verification based on Reachability Logic and symbolic execution (Section 4.2).

The closest approach to the one presented in this dissertation is [98]. Reachability Logic (presented in detail in Section 2.4) was introduced as an alternative to Floyd/Hoare logics and it is based on the operational semantics of a programming language. It consists in a language independent proof system (Figure 2.1) designed for program verification. It includes a special rule called Circularity which is primarily used to handle circular behaviours in programs, due to loops, recursivity, jumps, etc. In Reachability Logic, program properties are expressed using *reachability rules* of the form $\varphi \Rightarrow \varphi'$, where φ and φ' are called *patterns*, i.e. program configurations (with variables) constrained by a first-order formula.

In [92] the authors show that the proof system of Reachability Logic without the Circularity rule is *sound* and *complete*, i.e. any program property derived using the proof rules corresponds to an operational behaviour, and any operational behaviour can be derived using the rules from the proof system. In fact, the soundness result is the equivalent of our *Coverage* property, while the completeness result is a weak form of *Precision*. The proof system of Reachability Logic without Circularity captures symbolic execution, but at a high level of abstraction. In this dissertation,

we formalise symbolic execution using unification and then we show how we implement it using standard rewriting. Here, we make a clear distinction between code and data (integers, booleans, ...) in programs, and we only allow data to be symbolic. Our intent was to bring symbolic execution closer to the implementation, by formalising all its details. By doing so, we filled the gap between the (abstract) symbolic execution captured by the Reachability Logic proof system and its implementation based on rewriting. As a consequence, we were able to develop a (language independent) prototype for verifying programs using Reachability Logic. Our tool proves (or disproves) program properties given directly as reachability formulas, and we used it to verify non-trivial programs (Section 5.3).

An implementation closely related to ours is MatchC [93], which has beed used for verifying several challenging C programs such as the Schorr-Waite garbage collector. MatchC uses the Reachability Logic formalism for program specifications; it is, however, dedicated to a specific programming language, and uses a particular implementation of the RL proof system. By contrast, we focus on genericity, i.e., on language-independence: given a programming language defined in an algebraic/rewriting setting, we automatically generate the semantics for performing symbolic execution on that language. We propose our own proof system and a default program-verification strategy on the resulting symbolic execution engine. The soundness of our approach has also been proved. It relies on a Circularity Principle adapted to Reachability Logic, which has been formulated in a different setting in [95].

Other approaches offer support for verification of code contracts over programs. Spec# [15] is a tool developed at Microsoft that extends C# with constructs for non-null types, preconditions, postconditions, and object invariants. Spec# comes with a sound programming methodology that permits specification and reasoning about object invariants even in the presence of callbacks and multi-threading. A similar approach, which provides functionality for checking the correctness of a JAVA implementation with respect to a given UML/OCL specification, is the KeY [1] tool. In particular, KeY allows to prove that after running a method, its postcondition and the class invariant hold, using Dynamic Logic [52] and symbolic execution. The VeriFast tool [59] supports verification of single and multi-threaded C and Java programs annotated with preconditions and

postconditions written in Separation Logic [88]. An advanced tool used for verifying C programs is FramaC [32], an extensible and collaborative platform which allows to verify that the source code is compliant with a given specification. For functional specifications FramaC uses a dedicated language, ACSL (ANSI/ISO C Specification Language) [16]. The specifications can be partial, concentrating on one aspect of the analysed program at a time.

All these tools are designed to verify programs that belong to a specific programming language. Regarding performance, our generic tool for verification is (understandably) not in the same league as tools targeting specific languages and/or specific program properties. We believe, however, that the building of fast language-specific verification tools can benefit from the general principles presented in this dissertation, in particular, regarding the building of program-verification tools on top of symbolic execution engines.

1.4 Acknowledgements

Since before I started my PhD, when I was a master student, my research activity has been guided by Professor Dorel Lucanu. Thanks to him I was able to develop myself as a researcher, rather than being only a blind programmer. I would like to especially thank him for his efforts and countless hours he spent for every single issue that I had. He also deserves the credits for creating and managing a research group which was essential for my formation as a member. His contributions to the $\mathbb{K}$ framework and his inspiring ideas on symbolic execution played a major role in the development of this dissertation.

An important contribution to the development of this dissertation was made by Vlad Rusu. I would also like to especially thank him for the fruitful collaboration that we have.

During my master studies I published my first two papers under the supervision of prof. Adrian Iftene. I want to thank him for introducing me to the wonderful world of research and further support during my thesis.

Since before I started my PhD I am a member of the Formal Methods on Software Engineering group, which has a strong collaboration with the Formal Systems Laboratory group from UIUC. In this period, my former

and current colleagues have always been there to give me advices and ideas: Traian Florin Şerbănuţă, Radu Mereuţă, Ştefan Ciobâcă, Grigore Roşu, Chucky Ellison, Andrei Ştefănescu, Emilian Necula, Raluca Necula, Elena Naum, Mihai Asăvoae, Irina Măriuca Asăvoae. Thank you all!

Probably the key factor which helped me go through this period was my family. I would like to thank my parents, Gheorghiţă and Anişoara, and my sister, Andreea, for the fact that they have always been there for me. They have always trusted me and encouraged me with every choice that I have made. I would also want to especially thank my girlfriend, Andreea, for her patience, understanding, and support in the most difficult period of this PhD.

Last, but not least, I would like to thank my research committee, made up of Adrian Iftene, Marius Minea, Vlad Rusu, and Gheorghe Ştefănescu, for their patience and time spent to read and review this dissertation.

1.4.1 Grants

- In 2014, this work was supported by the strategic grant POSDRU /159/1.5/S/137750, "Project Doctoral and Postdoctoral programs support for increased competitiveness in Exact Sciences research" co-financed by the European Social Fund within the Sectorial Operational Program Human Resources Development 2007-2013
- Between 2011-2013, this work was supported by strategic grant POSCCE/161/15.06.2010, SMIS-CSNR 602-12516, "An Executable Semantic Framework for Rigorous Design, Analysis and Testing of Systems".

1.5 List of Publications

Below is the list of publications directly related to this dissertation:

1. *Andrei Arusoaie*, Dorel Lucanu and Vlad Rusu, *"A Generic Framework for Symbolic Execution"*, Proceedings of 6th International Con-

ference on Software Language Engineering (SLE'13), p. 260-281, LNCS, Indianapolis, USA, October, 2013

2. *Andrei Arusoaie*, *"Engineering Hoare Logic-based Program Verification in $\mathbb{K}$ Framework"*, Proceedings of SYNASC'13, p. 179-186, IEEE, Timişoara, Romania, September 23-26, 2013.

3. *Andrei Arusoaie*, Dorel Lucanu, Vlad Rusu, Traian Florin Şerbănuţă, Grigore Roşu and Andrei Ştefănescu, *"Language Definitions as Rewrite Theories"*, Proceedings of WRLA'14 , LNCS, Grenoble, France, april 5-6, 2014.

4. David Lazăr, *Andrei Arusoaie*, Traian Şerbănuţă, Chucky Ellison, Radu Mereuţă, Dorel Lucanu and Grigore Roşu, *"Executing Formal Semantics with the $\mathbb{K}$ Tool"*, Proceedings of FM'12, p. 267-271, LNCS, Paris, France, August 27-31, 2012.

5. *Andrei Arusoaie* and Daniel Ionuţ Vicol, *"Automating Abstract Syntax Tree construction for Context Free Grammars"*, Proceedings of SYNASC'12, p. 152-160, IEEE, Timişoara, Romania, September 26-29, 2012

6. *Andrei Arusoaie*, Traian Florin Şerbănuţă, Chucky Ellison and Grigore Roşu, *"Making Maude Definitions more Interactive"*, Proceedings of WRLA'12, p. 83-99, LNCS, Tallinn, Estonia, March 24-25, 2012

7. *Andrei Arusoaie*, Dorel Lucanu and Vlad Rusu, *"Towards a K semantics of OCL"*, ENTCS, 2014, p. 81-96

8. Traian Florin Şerbănuţă, *Andrei Arusoaie*, David Lazăr, Chucky Ellison, Dorel Lucanu and Grigore Roşu, *The $\mathbb{K}$ Primer (version 3.3)*, ENTCS, 2014, p. 57-80

1.6 Participation to Summer Schools, Workshops, Internships

Summer Schools:

1. *Summer School on Coq*, NII Shonan Meeting, 2014, Shonan, Kanagawa, Japan

2. International Summer School on *Software Systems Safety*, MOD 2013, Marktoberdorf, Germany

3. Summer School on Language Frameworks, SSLF 2012, Sinaia, Romania.

Workshops:

1. The 22^{nd} *International Workshop on Algebraic Development Techniques*, WADT, 2014, Sinaia, Romania

2. The 2^{nd} *International* $\mathbb{K}$ *workshop*, K'11, Cheile Grădiştei, Braşov, Romania

3. The 1^{st} *International Workshop on the* $\mathbb{K}$ *Framework and its Applications*, K'10, Nags Head, North Carolina, USA

Internships:

1. INRIA Lille Nord Europe, October 1^{st} - November 15^{th} 2011, France

2. INRIA Lille Nord Europe, July 1^{st} - July 30^{th} 2013, BQR grant, France

List of Figures

Chapter 2

Background

This chapter revisits some theoretical material used in this dissertation. The basics of algebraic specifications, together with the definitions and the notational conventions that we are going to use through the dissertation, are presented in Section 2.1. Section 2.2 contains a brief description of many-sorted First Order Logic. In Section 2.3 we present Matching Logic, a logic designed to state and reason about structural properties of arbitrary program configurations. Matching Logic formulas are the basic ingredients of Reachability Logic (Section 2.4), a logic used for defining the operational semantics of languages, and for stating program properties. At the end of the chapter, in Section 2.5, we present $\mathbb{K}$, a framework for defining formal semantics of programming languages. All these will be assumed known for the subsequent development of the dissertation.

2.1 Algebraic Specifications

In this section we briefly introduce, inspired from [47], the basic knowledge about algebraic specifications (definitions and notations) used through this dissertation. We assume that the reader is already familiar with set theory as it is used in mathematics today.

The use of algebraic specifications to model computer programs is motivated by the fact that they can manipulate several kinds of *sorts* of data, in the same way as programs do. Programs can be represented in terms of sets of data values and mathematical functions over these sets.

When using an algebra to model a program, there is a natural one-to-one correspondence between the sets of values in the algebra and the sorts of data in the program.

Let S be a set of sorts. We use S^* to denote the set of all lists of elements from S. For example, if $S = \{a, b, c, d\}$ then a, ab, abd, $[] \in S^*$, where $[]$ is the empty list. We often use w to denote non-empty lists of elements from S^* (e.g. $w = abd$).

A *map* is a function $f : A \to B$ with given source (domain) and target (codomain). In the dissertation, we make use of the following definitions and notations:

- The composition of two functions $f : A \to B$ and $g : B \to C$ is denoted by $g \circ f : A \to C$, where $(g \circ f)(x) = g(f(x))$ for all $x \in A$.
- Given two functions $f : A \to B$ and $g : A \to B$ we say that f and g are (extensionally) equal, denoted $f = g$, if for all $x \in A$ we have $f(x) = g(x)$.
- If $f : A \to B$ is a function and A' is a subset of A (i.e. $A' \subseteq A$) then the *restriction of f to A'* is the function $f|_{A'} : A' \to B$ defined as $f|_{A'}(x) = f(x)$, for all $x \in A'$.
- We use $1_A : A \to A$ to denote the identity function on set A: $1_A(x) = x$ for each $x \in A$.

Next, we define sets indexed by a set of sorts:

Definition 2.1.1 (S-indexed set) *Given S a set of sorts, an* S-indexed set *A is a set-valued map with source S and target $\{A_s \mid s \in S\}$, where the value at $s \in S$ is denoted by A_s.*

By $|A|$ we denote $\bigcup_{s \in S} A_s$ and we let $a \in A$ mean that $a \in |A|$. The *empty* S-indexed set has $\emptyset_s = \emptyset$ for each sort s in S, and we (ambiguously) denoted it by $\emptyset$. If A and B are two S-indexed sets then:

- $A \subseteq B$ iff $A_s \subseteq B_s$, for each $s \in S$, and
- $A \cup B \triangleq \{C_s \mid C_s = A_s \cup B_s, s \in S\}$.

Thus, symbols $\subseteq$ and $\cup$, which correspond to inclusion and union operations are overloaded for S-indexed sets. The other known operations on sets, namely intersection ($\cap$), set difference ($\setminus$), and cartesian product ($\times$) are similarly extended to S-indexed sets.

In general, S-indexed sets allow us to manipulate a family of sets as a unit, in such a way that operations on this unit respect the "typing" of data values. We provide examples of S-indexed sets in Example 2.1.2.

The syntax of a programming language consists in a set of rules which specify how different symbols can be combined to obtain a (syntactically) correct program. It includes the set of data entities (sorts), the operations on them together with the sorts of their arguments, and the return sort. In an algebraic setting the syntax is given using *signatures*.

Definition 2.1.2 (Many-sorted signature) *Given S a set of sorts, an* S-sorted signature Σ *is an $S^* \times S$-indexed family of sets $\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$ of sets whose elements are called* operation symbols. *An operation symbol $f \in \Sigma_{\omega,s}$ is said to have* arity w *and sort s. A symbol $c \in \Sigma_{[],s}$ is called a* constant *symbol.*

In general, the syntax of a programming language is specified using a Backus-Naur Form [75] (abbreviated BNF) grammar, which consists of specification rules of the form

$$\mathit{Symbol} ::= \mathit{seq_1} \mid \mathit{seq_2} \mid \ldots,$$

where *Symbol* is a *nonterminal* and $\mathit{seq_1}$, $\mathit{seq_2}$, ... are sequences of *terminals* (symbols that never appear in the left side of ::=) or nonterminals; these sequences are separated by the | operator which indicates a *choice* between sequences, i.e. each such sequence can be a possible substitution for *Symbol*. We often refer to these sequences as *syntax productions*.

In Example 2.1.1 we show the BNF syntax of a simple language which deals only with expressions. *Int*, *Bool*, and *Exp* are nonterminals, while $\underline{0}$, $\underline{\mathit{true}}$ are terminal symbols, and *Int*, *Exp* + *Exp* are syntax production corresponding to nonterminal *Exp*. The choice operator "|" inside $\underline{\mathit{true}} \mid \underline{\mathit{false}}$ specifies the fact that *Bool* can be either the $\underline{\mathit{true}}$ or $\underline{\mathit{false}}$.

Example 2.1.1 *Here we show the syntax of a very simple language which only deals with integer and boolean expressions:*

$$
\begin{array}{lcl}
\mathit{Bool} & ::= & \underline{\mathit{true}} \mid \underline{\mathit{false}} \\
\mathit{Int} & ::= & \ldots \mid \underline{-1} \mid \underline{0} \mid \underline{1} \mid \ldots \\
\mathit{Exp} & ::= & \mathit{Int} \mid \mathit{Bool} \\
 & \mid & \mathit{Exp} * \mathit{Exp} \\
 & \mid & \mathit{Exp} \,/\, \mathit{Exp} \\
 & \mid & \mathit{Exp} + \mathit{Exp} \\
 & \mid & \mathit{Exp} - \mathit{Exp} \\
 & \mid & \neg \mathit{Exp} \\
 & \mid & \mathit{Exp} \wedge \mathit{Exp} \\
 & \mid & (\ \mathit{Exp}\ )
\end{array}
$$

The grammar above includes the non-terminals Bool, Int, and Exp, and the terminals ($*$*, /, +, -, (,), ¬, ∧), $\underline{\mathit{true}}$, $\underline{\mathit{false}}$, $\underline{-1}$, $\underline{0}$, $\underline{1}$, ...).*

The BNF grammar of a language has a corresponding S-sorted signature Σ. For each syntax production of the form $\mathit{Symbol} ::= \mathit{seq} \mid \ldots$, the nonterminal Symbol is a sort in S, and Σ includes the indexed sets obtained as below:

- If $\mathit{seq} \triangleq e$, where e is a terminal symbol, then $t \in \Sigma_{[],\mathit{Symbol}}$.
- If $\mathit{seq} \triangleq e$, where e is a nonterminal symbol, then $\Sigma_{e,\mathit{Symbol}}$ includes an operation symbol $\iota_{e,\mathit{Symbol}}$.
- If $\mathit{seq} \triangleq e_1 e_2 \ldots e_n$ then the concatenation $e \triangleq t(e_1)t(e_2)\ldots t(e_n) \in \Sigma_{w,\mathit{Symbol}}$; t is the function which maps e_i to symbol _ (if e_i is a nonterminal) or to e_i (if e_i is a terminal), for all $i \in \{1, 2, \ldots, n\}$; w is obtained by concatenating the sorts of e_i.

In Example 2.1.2 we show the corresponding many sorted signature of the BNF grammar from Example 2.1.1.

Example 2.1.2 *The set S includes the nonterminals Bool, Int, and Exp as sorts, that is,* $S = \{\mathit{Bool}, \mathit{Int}, \mathit{Exp}\}$. *For each syntax production we generate the operation names by concatenating the symbol _ (corresponding to nonterminal symbols) and the terminal symbols in the same order they*

appear in the production (e.g. for production Exp + Exp we generate the operation name _+_).

The following indexed sets:

- $\Sigma_{[],Bool} = \{\underline{true}, \underline{false}\}$,
- $\Sigma_{Bool,Bool} = \{\neg_{Bool}\_\}$
- $\Sigma_{Bool\ Bool,Bool} = \{\_ \wedge_{Bool} \_, \_ \vee_{Bool} \_, \ldots\}$
- $\Sigma_{[],Int} = \{\ldots, \underline{-1}, \underline{0}, \underline{1}, \ldots\}$,
- $\Sigma_{Int\ Int,Int} = \{\_ +_{Int} \_, \_ -_{Int} \_, \_ *_{Int} \_, \_ /_{Int} \_, \ldots\}$
- $\Sigma_{Exp,Exp} = \{\neg\_, (\_)\}$, *and*
- $\Sigma_{Exp\ Exp,Exp} = \{\_+\_, \_-\_, \_*\_, \_/\_, \_ \wedge \_\}$
- $\Sigma_{Int,Exp} = \{\iota_{Int,Exp}\}$.
- $\Sigma_{Bool,Exp} = \{\iota_{Bool,Exp}\}$.

are included in the S-sorted signature Σ, which corresponds to the syntax of the language shown in Example 2.1.1.

There are cases when we may want to refer to a specific part of the signature corresponding to a language, for instance, the one which includes only the *data* sorts in the language (e.g. integers, booleans, ...). We call that part a *subsignature*:

Definition 2.1.3 (Subsignature) *Given S and S' two sets of sorts such that $S' \subseteq S$, an S'-sorted signature Σ' is a* subsignature *of an S-sorted signature Σ if $\Sigma' \subseteq \Sigma$ as $S^* \times S$-indexed sets.*

Remark 2.1.1 *The subsignature Σ' can be regarded as an S-sorted signature where $\Sigma'_{w,s} = \emptyset$ for any $(w, s) \in (S^* \times S) \setminus (S'^* \times S')$.*

Example 2.1.3 *Recall the simple language from Example 2.1.2 and its corresponding signature Σ. The signature Σ' containing only the indexed sets:*

- $\Sigma_{[],Bool} = \{\underline{true}, \underline{false}\}$ *and*
- $\Sigma_{Bool,Bool} = \{\neg_{Bool}\_\}$
- $\Sigma_{Bool\ Bool,Bool} = \{\_ \wedge_{Bool} \_, \_ \vee_{Bool} \_, \ldots\}$

is a subsignature of Σ.

Signatures specify the syntax of programming languages but we are also interested in their semantics, that is, the entities of different sorts and particular functions that interpret the function symbols from signatures. For this we define Σ-algebras:

Definition 2.1.4 (Σ-algebra) *Given* Σ *an* S*-sorted signature, a* Σ*-algebra* M *consists of an* S*-indexed set (also denoted* M*), i.e., a* carrier set M_s *for each sort* $s \in S$*, plus:*

- *an element* $M_c \in M_s$ *for each* $c \in \Sigma_{[],s}$ *interpreting the constant symbol* c *as an actual element, and*
- *a function* $M_f : M_{s_1} \times \ldots M_{s_n} \to M_s$ *for each* $f \in \Sigma_{w,s}$*, where* $w = s_1 \ldots s_n$ *(*$n > 0$*), interpreting each operation symbol as a function.*

A Σ-algebra interprets every language construct (symbol in signature Σ) either as a constant or as a function. In this dissertation we assume that the carrier sets M_s are disjoint.

Example 2.1.4 *Consider the many sorted signature* Σ *shown in Example 2.1.1. A* Σ*-algebra* M *includes the carrier sets* M_{Int} *(which is in fact the set of integers),* M_{Bool} *(the set of boolean values, i.e. true and false), and* M_{Exp}*. For every constant symbol we have a corresponding element in one these carrier sets, e.g.* $M_{\underline{1}} = 1 \in M_{Int}$*. We always interpret operation symbols* $\iota_{Sort',Sort}$ *as injections* $M_{Sort'} \to M_{Sort}$*. For instance, we have the injection* $M_{\iota_{Int,Exp}} : M_{Int} \to M_{Exp}$*. We often write* I *for* $M_{\iota_{Int,Exp}}(I)$ *whenever the injection is deduced from the context. The algebra* M *also contains functions interpreting all operation symbols, i.e. the symbol* $\_ + \_$ *is interpreted by a function* $\_ +_{Int} \_ : Int \times Int \to Int$ *which is the addition over integers.*

The terms over a signature Σ form a special algebra, called term algebra. In the following, we define the set of Σ-terms with variables, and the corresponding algebra. The set of variables, denoted *Var*, is an infinite S-indexed set of symbols disjoint from Σ.

Definition 2.1.5 (Σ-terms) *Let Σ be an S-sorted signature and Var an S-indexed set of* variables *disjoint from Σ. The S-indexed set $T_\Sigma(Var) = \{T_{\Sigma,s}(Var) \mid s \in S\}$ of* Σ-terms *t is defined by:*

$$t ::= c \mid X \mid f\underline{(}t, \ldots, t\underline{)},$$

where c ranges over constant symbols from $\Sigma_{[],s}$, X ranges over variables from Var_s, and f over operation symbols from $\Sigma_{s_1 \ldots s_n, s}$.

Definition 2.1.6 (Term algebra) *Given an S-sorted signature Σ and an S-indexed set of* variables *Var, the* term Σ-algebra *$T_\Sigma(Var)$ has Σ-terms $T_\Sigma(Var)$ as carrier sets and interprets each constant symbol c by itself and each operation symbol $f : s_1 \ldots s_n \to s$ by the function $T_{s_1} \times \ldots \times T_{s_n} \to T_s$ that maps $(t_1, \ldots, t_n)$ into the term $f\underline{(}t_1, \ldots, t_n\underline{)}$.*

Example 2.1.5 *Let Σ be the signature shown in Example 2.1.2. Then true, $\_ + \_\underline{(}2, 3\underline{)}$, and $\_ * \_\underline{(}\_ + \_\underline{(}2, 3\underline{)}, x\underline{)}$, where $X \in Var_{Exp}$, are all Σ-terms interpreted as themselves, i.e. true, $\_ + \_\underline{(}2, 3\underline{)}$, and $\_ * \_\underline{(}\_ + \_\underline{(}2, 3\underline{)}, X\underline{)}$ respectively, in the corresponding term algebra.*

We often use only (and) instead of $\underline{(}$ and $\underline{)}$, respectively. The Σ-algebra T_Σ of *ground* Σ-terms is the Σ-algebra $T_\Sigma(\overline{\emptyset})$ of terms without variables.

The *subterms* of a term can be referred using *positions*, which are lists of integers. The *subterm of term t at position p* is denoted by $t|_p$, and is defined inductively as follows:

- $t|_{[]} = t$, where [] is the empty list, and
- $f(t_1, \ldots, t_n)|_{iq} = t_i|_q$, if $1 \leq i \leq n$.

Note that constants and variables do not have proper subterms. We denote by $t[u]_p$ the term obtained from t by *replacing the subterm at position p by u*, i.e.:

- $t[u]_{[]} = u$, and
- $f(t_1, \ldots, t_n)[u]_{iq} = f(t_1, \ldots, t_i[u]_q, \ldots, t_n)$, if $1 \leq i \leq n$.

Example 2.1.6 *Let* $t \triangleq \_ * \_(\_ + \_(Y, 3), X)$. *Then,* $t|_{[]} = t$, $t|_{1\,2} = 3$, $t[Z]_{[]} = Z$, *and* $t[Z]_2 = \_ * \_(\_ + \_(Y, 3), Z)$.

By $vars(t)$ we denote the set of variables occurring in the term t, $vars : T_\Sigma(Var) \to 2^{Var}$ being defined as follows:

- $vars(X) = \{X\}$, $X \in Var$,
- $vars(c) = \emptyset$, c is a constant, and
- $vars(f(t_1, \ldots, t_n)) = vars(t_1) \cup \ldots \cup vars(t_n)$.

For terms $t_1, \ldots, t_n$ we let $vars(t_1, \ldots, t_n) \triangleq vars(t_1) \cup \cdots \cup vars(t_n)$.

Example 2.1.7 *If* $t \triangleq \_ * \_(\_ + \_(Y, 3), X)$ *then* $vars(t) = \{X, Y\}$.

Let M be a Σ-algebra. A *valuation* ρ is a function $\rho : Var \to M$ which maps variables to values from M. Any valuation ρ can be extended to $\rho^\# : T_\Sigma(Var) \to M$ as follows:

- $\rho^\#(X) = \rho(X)$, $X \in Var$,
- $\rho^\#(c) = M_c$, and
- $\rho^\#(f(t_1, \ldots, t_n)) = M_f(\rho^\#(t_1), \ldots, \rho^\#(t_n))$.

For simplicity, in the rest of the dissertation, we use ρ instead of its extension $\rho^\#$.

Example 2.1.8 *Let* $t \triangleq * (+ (Y, Z), X)$ *and* $\rho : Var \to M$ *a valuation such that* $\rho(X) = 1$, $\rho(Y) = 2$, *and* $\rho(Z) = 3$, *where* $\{1, 2, 3\} \subseteq M_{Int}$. *Then,* $\rho(t) \triangleq \rho^\#(t) = M_*(M_+(2, 3), 1)$.

Given two valuations $\rho : V \to M$ and $\rho' : V' \to M$, where $V, V' \subseteq Var$, such that for any $X \in V \cap V'$ we have $\rho(X) = \rho'(X)$, we define the valuation $\rho \uplus \rho' : V \cup V' \to M$ as

$$(\rho \uplus \rho')(X) = \begin{cases} \rho(X) & , X \in V, \\ \rho'(X) & , X \in V'. \end{cases}$$

Note that the valuation $\rho \uplus \rho'$ is well defined since $\rho(X) = \rho'(X)$ for all $X \in V \cap V'$.

A *substitution* is a valuation $\sigma : Var \to T_\Sigma(Var)$, i.e., a valuation where M is the algebra of Σ-terms with variables. The identity substitution is denoted $1_V : V \to V$, and $1_V(X) = X$, for all $X \in V (\subseteq Var)$.

2.2 Many-Sorted First Order Logic

This section contains an overview of many-sorted First Order Logic, abbreviated FOL in this dissertation. FOL is a formal system which provides a mechanism to express properties about objects together with their logical relationships and dependences.

The basic ingredients of the FOL language are terms, which can be constants, variables or complex terms built by applying functions to other terms. In addition to having terms, FOL contains a set of predicate symbols and quantifiers. Many-sorted FOL is obtained by adding to the FOL formalism the notion of *sort*. Thus, in contrast to usual (unsorted) FOL, the arguments of functional and predicate symbols may have different sorts, and constant and functional symbols also have particular sorts.

In the following we formally define the syntax of many-sorted FOL:

Definition 2.2.1 (Many-Sorted First-Order Signature) *Given S a set of sorts, an S-sorted* first order signature Φ *is a pair (Σ, Π), where Σ is an S-sorted signature and Π is an indexed set of the form $\{\Pi_w \mid w \in S^*\}$ whose elements are called* predicate symbols*, where $\pi \in \Pi_w$ is said to have* arity *w.*

Next, we define the syntax of FOL formulas over a first order signature $\Phi = (\Sigma, \Pi)$. We let S denote the set of sorts in Φ and Var an infinite S-indexed set of variable symbols (disjoint from Σ).

Definition 2.2.2 (FOL formula) *The set of* Φ-formulas *is defined by*

$$\phi ::= \top \mid p(t_1, \ldots, t_n) \mid \neg\phi \mid \phi \wedge \phi \mid (\exists V)\phi$$

where p *ranges over predicate symbols* Π*, each* t_i *ranges over* $T_\Sigma(Var)$ *of appropriate sort, and* V *over finite subsets of* Var*.*

The other known connectives, $\vee$, $\rightarrow$, and the quantifier $\forall$ can be expressed using the ones above:

- $\phi \vee \phi' \triangleq \neg(\neg\phi \wedge \neg\phi')$,
- $\phi \rightarrow \phi' \triangleq \neg\phi \vee \phi'$, and
- $(\forall V)\phi \triangleq \neg(\exists V)\neg\phi$

For convenience, we often use parenthesis for disambiguation or scoping. The syntactical constructs have the following priorities, in decreasing order, starting with the ones which bind tighter:

1. $\neg$, $\exists$, $\forall$,
2. $\wedge$, $\vee$,
3. $\rightarrow$.

In a FOL formula a variable may occur *free* or *bound*.

Definition 2.2.3 (Free variables) *A variable* X *is* free *in a FOL formula* ϕ *if:*

- $\phi \triangleq \top$*;*
- $\phi \triangleq p(t_1, \ldots, t_n)$*;*
- $\phi \triangleq \neg\psi$ *and* X *is free in the FOL formula* ψ*;*
- $\phi \triangleq \psi_1 \wedge \psi_2$ *and* X *is free in both FOL formulas* ψ_1 *and* ψ_2*;*
- $\phi \triangleq (\exists V)\psi$ *and* $X \notin V$ *and* X *is free in the FOL formula* ψ*.*

An occurrence of a variable is bound in a formula if it is in the scope of a quantifier.

Definition 2.2.4 (Bound variables) *A variable X is* bound *in a FOL formula ϕ if $\phi \triangleq (\exists V)\psi$ and $X \in V$.*

Example 2.2.1 *In the formula $(\forall X)(X \wedge Y \vee \pi(Z))$ the variable X is bound, while Y and Z are free.*

Definition 2.2.5 *A substitution $\sigma : Var \to T_\Sigma(Var)$ can be extended to FOL formulas:*

- $\sigma^\#(\top) = \top$;
- $\sigma^\#(p(t_1, \ldots, t_n)) = p(\sigma^\#(t_1), \ldots, \sigma^\#(t_n))$;
- $\sigma^\#(\neg\phi) = \neg\sigma^\#(\phi)$;
- $\sigma^\#(\phi_1 \wedge \phi_2) = \sigma^\#(\phi_1) \wedge \sigma^\#(\phi_2)$;
- $\sigma^\#((\exists V)\phi) = (\exists V)\sigma'^\#(\phi)$,

where $\sigma' : Var \to T_\Sigma(Var)$, $\sigma'(Y) = Y$ when $Y \in V$, and $\sigma'(Y) = \sigma(Y)$ when $Y \notin V$.

Notice that a substitution σ automatically avoids substituting for bound variables.

Example 2.2.2 *If σ is a substitution, where $\sigma(X) = Y$, $\sigma(Y) = Z$ then $\sigma((\forall X)(X \wedge \pi(Y))) = (\forall X)(X \wedge \pi(Z))$.*

The set of free variables occurring in a FOL formula ϕ, denoted $vars(\phi)$, is defined below:

- $vars(\top) = \emptyset$;
- $vars(p(t_1, \ldots, t_n)) = vars(t_1) \cup \ldots \cup vars(t_n)$;
- $vars(\neg\phi) = vars(\phi)$;
- $vars(\phi_1 \wedge \phi_2) = vars(\phi_1) \cup vars(\phi_2)$;
- $vars((\exists V)\phi) = vars(\phi) \setminus V$.

Example 2.2.3 *Given the FOL formula* $F \triangleq (\forall X)(X \wedge Y \vee \pi(Z))$ *we have* $vars(F) = \{Y, Z\}$.

The truth value of a FOL formula depends on the choice of values for variables and the meaning of the operation and predicate symbols involved. Thus, we require a *model* of all operations and predicates.

Definition 2.2.6 (FOL Model) *Given a FOL signature* $\Phi = (\Sigma, \Pi)$*, a* Φ-model *consists of a* Σ*-algebra* M *together with a subset* $M_p \subseteq M_{s_1} \times \cdots \times M_{s_n}$ *for each predicate* $p \in \Pi_w$*, where* $w = s_1 \ldots s_n$.

Definition 2.2.7 (FOL Satisfaction relation) *Given a first order* Φ*-model* M*,a* Φ*-formula* ϕ*, and a valuation* $\rho : Var \to M$*, the satisfaction relation* $\rho \models_{\text{FOL}} \phi$ *is defined as follows:*

1. $\rho \models_{\text{FOL}} \top$*;*
2. $\rho \models_{\text{FOL}} p(t_1, \ldots, t_n)$ *iff* $(\rho(t_1), \ldots, \rho(t_n)) \in M_p$*;*
3. $\rho \models_{\text{FOL}} \neg\phi$ *iff* $\rho \models_{\text{FOL}} \phi$ *does not hold;*
4. $\rho \models_{\text{FOL}} \phi_1 \wedge \phi_2$ *iff* $\rho \models_{\text{FOL}} \phi_1$ *and* $\rho \models_{\text{FOL}} \phi_2$*;*
5. $\rho \models_{\text{FOL}} (\exists V)\phi$ *iff there is* $\rho' : Var \to M$ *with* $\rho'(X) = \rho(X)$*, for all* $X \notin V$*, such that* $\rho' \models_{\text{FOL}} \phi$.

A formula ϕ *is* valid in M*, denoted by* $M \models_{\text{FOL}} \phi$*, if it is satisfied by all valuations* ρ.

When the model M is clearly understood from the context we only write $\models_{\text{FOL}} \phi$ instead of $M \models_{\text{FOL}} \phi$, to denote that ϕ is valid in M. As a remark, in this dissertation we do not use $\models_{\text{FOL}} \phi$ as notation for universally valid formulas since we work with a fixed model M.

2.3 Matching Logic

Matching Logic (ML) was first introduced in [94] as a novel framework for defining axiomatic semantics for programming languages. In general, logics designed for program reasoning do not concern about the low-level

details of program configurations. In contrast, ML has the program configuration at its core, capturing information about the structure of the various data in a program's state. ML specifications are symbolic program configurations constrained by a FOL formula, called *patterns*. A concrete configuration satisfies a pattern if it *matches* the symbolic configuration of the pattern and satisfies the FOL constraint. We recall here the syntax and the semantics of Matching Logic as presented in [98].

Definition 2.3.1 (ML Signature) *An ML signature* $\Phi = (\Sigma, \Pi, \mathit{Cfg})$ *is a first-order signature* (Σ, Π) *together with a distinguished sort Cfg for* configurations.

The sort *Cfg* is intended to model program configurations. The configuration terms may contain informations about the program state (the heap, the stack, the input, the output, etc.) and appear in ML formulas.

Example 2.3.1 *Let us consider a simple program configuration which is a pair* $\langle\langle \texttt{code} \rangle \langle m \rangle\rangle$ *with* `code` *a fragment of program and* m *a mapping from program variables to integers. Then,* $\langle\langle \texttt{x = 2;} \rangle \langle \texttt{x} \mapsto 5 \rangle\rangle$ *and* $\langle\langle \texttt{if x <= 2 then \{x = 0; \}} \rangle \langle \texttt{x} \mapsto 1 \rangle\rangle$ *are terms of sort Cfg.*

In the rest of the section, we consider an ML-signature $\Phi = (\Sigma, \Pi, \mathit{Cfg})$, and we let *Var* be an S-indexed set of variables (disjoint from Σ). We now define the syntax of ML formulas over the ML-signature Φ.

Definition 2.3.2 (ML formula) *The set of* ML-formulas *is defined by*

$$\varphi ::= \pi \mid \top \mid p(t_1, \ldots, t_n) \mid \neg\varphi \mid \varphi \wedge \varphi \mid (\exists V)\varphi$$

where π *ranges over* $T_{\Sigma, \mathit{Cfg}}(\mathit{Var})$*,* p *ranges over predicate symbols* Π*, each* t_i *ranges over* $T_\Sigma(\mathit{Var})$ *of appropriate sorts, and* V *over finite subsets of Var.*

The syntax of ML formulas is quite similar with the syntax of FOL formulas, except that the former allows configuration terms $\pi \in T_{\Sigma, \mathit{Cfg}}(\mathit{Var})$ to be also formulas. The ML formulas can be encoded in FOL [98] (Definition 2.3.4), but in the dissertation we prefer to give a direct semantics. In addition to the above syntax we also allow $\vee$, $\rightarrow$, and $\forall$ in ML formulas,

which can be expressed in terms of the existent ones in the same way as we did for FOL (see Section 2.2).

We distinguish the two particular types of ML formulas: *basic patterns* and (elementary) *patterns*. A *basic pattern* is a term $\pi \in T_{\Sigma,Cfg}(Var)$. A *pattern* is an ML formula of the form $\pi \wedge \phi$, where π is a basic pattern and ϕ is a FOL formula. The basic pattern π is a configuration term with variables, and thus, it defines a set of (concrete) configurations. The condition ϕ gives additional constraints these configurations must satisfy.

Example 2.3.2 *Consider the program configuration* $\langle\langle \texttt{code} \rangle \langle m \rangle\rangle$ *shown in Example 2.3.1. Then,* $\pi \triangleq \langle\langle \texttt{if x > 0 \{ x = x + 1; \}} \rangle \langle \texttt{x} \mapsto X \rangle\rangle$ *is a basic pattern, where* `x` *is a program variable and* X *is a logical variable, and* $\pi \wedge \phi \triangleq \langle\langle \texttt{if x > 0 \{ x = x + 1; \}} \rangle \langle \texttt{x} \mapsto X \rangle\rangle \wedge X > 0$ *is a pattern, which consists of a basic pattern plus an additional constraint over* X*.*

We now define the ML satisfaction relation.

Definition 2.3.3 (ML satisfaction relation) *Given an ML signature* $\Phi = (\Sigma, \Pi, Cfg)$*,* M *a* (Σ, Π)*-model,* ϕ *a ML formula,* $\gamma \in M_{Cfg}$ *a concrete configuration, and* $\rho : Var \to M$ *a valuation, then the satisfaction relation* $(\gamma, \rho) \models_{\text{ML}} \varphi$ *is defined as follows:*

1. $(\gamma, \rho) \models_{\text{ML}} \pi$ *iff* $\rho(\pi) = \gamma$*;*
2. $(\gamma, \rho) \models_{\text{ML}} \top$*;*
3. $(\gamma, \rho) \models_{\text{ML}} p(t_1, \ldots, t_n)$ *iff* $(\rho(t_1), \ldots, \rho(t_n)) \in M_p$*;*
4. $(\gamma, \rho) \models_{\text{ML}} \neg\varphi$ *iff* $(\gamma, \rho) \models_{\text{ML}} \varphi$ *does not hold;*
5. $(\gamma, \rho) \models_{\text{ML}} \varphi_1 \wedge \varphi_2$ *iff* $(\gamma, \rho) \models_{\text{ML}} \varphi_1$ *and* $(\gamma, \rho) \models_{\text{ML}} \varphi_2$*; and*
6. $(\gamma, \rho) \models_{\text{ML}} (\exists V)\varphi$ *iff there is* $\rho' : Var \to M$ *with* $\rho'(X) = \rho(X)$*, for all* $X \notin V$*, such that* $(\gamma, \rho') \models_{\text{ML}} \varphi$*.*

An ML formula ϕ *is* valid*, written* $\models_{\text{ML}} \varphi$*, when* $(\gamma, \rho) \models \varphi$ *for all* $\gamma \in T_{\Sigma,Cfg}$ *and all* $\rho : Var \to M$*.*

In ML, a basic pattern π is satisfied by (γ, ρ), where γ is a configuration term and ρ a valuation, if $\rho(\pi) = \gamma$. The valuation ρ actually gives a possible *match* of configuration γ against basic pattern π, wherefrom the name of the logic: Matching Logic.

Example 2.3.3 *Consider the program configuration* $\langle\langle \texttt{code} \rangle \langle m \rangle\rangle$ *shown in Example 2.3.1. If* $\pi \triangleq \langle\langle C \rangle \langle \texttt{x} \mapsto X \rangle\rangle$ *is a basic pattern (*C *and* X *are variables), then the configuration* $\gamma \triangleq \langle\langle \texttt{skip} \rangle \langle \texttt{x} \mapsto 7 \rangle\rangle$ *matches against* π *since there exists* $\rho : Var \to M$*, where* $\rho(X) = 7$ *and* $\rho(C) = \texttt{skip}$*, such that* $\gamma = \rho(\pi)$*. Moreover,* γ *also matches against* $\pi \wedge \phi \triangleq \langle\langle C \rangle \langle \texttt{x} \mapsto X \rangle\rangle \wedge X \geq 0$ *using the same valuation* ρ*, since* $\gamma = \rho(\pi)$*, and* $\rho \models_{\text{FOL}} \phi$*, since* $7 \geq 0$*.*

Next, we recall from [98] how pattern reasoning in ML reduces to FOL reasoning in the (Σ, Π)-model M:

Definition 2.3.4 (FOL encoding of ML) *Let* $\square$ *be a special fresh Cfg variable, such that* $\square \notin Var$*, and* $Var^{\square} = Var \cup \{\square\}$*. For a pattern* φ*, let* $\varphi^{\square}$ *be the FOL formula obtained by replacing basic patterns* $\pi \in T_{\Sigma, Cfg}(Var)$ *with equalities* $\pi = \square$*. If* $\gamma \in M_{Cfg}$ *and* $\rho : Var \to M$*, then let* $\rho^{\gamma} : Var^{\square} \to M$ *be a valuation, such that* $\rho^{\gamma}(X) = \rho(X)$ *for all* $X \in Var$ *and* $\rho^{\gamma}(\square) = \gamma$*.*

Using the notation from Definition 2.3.4, we have $(\gamma, \rho) \models_{\text{ML}} \varphi$ iff $\rho^{\gamma} \models_{\text{FOL}} \varphi^{\square}$, which is proved by structural induction on φ (the complete proof can be found in [99]). This result allows us to extend all the constructions over FOL formulas to ML formulas.

The set of concrete configurations that match against a specific pattern is defined below.

Definition 2.3.5 *If* φ *is a ML formula, then* $[\![\varphi]\!]$ *denotes the set of concrete configurations* $\{\gamma \mid \text{there exists } \rho \text{ s.t. } (\gamma, \rho) \models_{\text{ML}} \varphi\}$*. For any set of ML formulas* F *we let* $[\![F]\!]$ *denote the set* $\bigcup_{\varphi \in F} [\![\varphi]\!]$*.*

Remark 2.3.1 *Let us consider* $\varphi \triangleq \langle\langle \texttt{if } B \texttt{ \{ } S \texttt{ \}} \rangle \langle E \rangle\rangle \wedge (B = true)$ *and* $\varphi' \triangleq (\exists B) \langle\langle \texttt{if } B \texttt{ \{ } S \texttt{ \}} \rangle \langle E \rangle\rangle \wedge (B = true)$ *two ML formulas, with* B*,* S*, and* E *from* Var*. We can observe that* $[\![\varphi]\!] = [\![\varphi']\!]$*, since* $\gamma \in [\![\varphi]\!]$ *obviously implies* $\gamma \in [\![\varphi']\!]$*, and* $\gamma \in [\![\varphi']\!]$ *implies* $\gamma \in [\![\varphi]\!]$ *(by taking* $\rho(B) = true$*). Thus, by the semantics of* $[\![\_]\!]$*,* φ *and* φ' *are equivalent. However,* $\varphi^{\square}$ *and* $\varphi'^{\square}$ *are not equivalent as FOL formulas, because of the existential quantifier in* $\varphi'^{\square}$ *(i.e. consider a valuation* $\rho : Var \to M$*, such that* $\rho(B) = false$ *and* $\rho \models_{\text{FOL}} \varphi'^{\square}$*; then* $\rho \not\models_{\text{FOL}} \varphi^{\square}$*).*

The next lemma shows that a formula $(\exists V)\varphi$, $V \subseteq vars(\varphi)$, matches the same set of concrete configurations as φ.

Lemma 2.3.1 *Let φ be an ML formula. Then $[\![(\exists V)\varphi]\!] = [\![\varphi]\!]$.*

Proof We prove $[\![(\exists V)\varphi]\!] \subseteq [\![\varphi]\!]$ and $[\![(\exists V)\varphi]\!] \supseteq [\![\varphi]\!]$:

($\subseteq$) Let $\gamma \in [\![(\exists V)\varphi]\!]$. By Definition 2.3.5, there exists $\rho : Var \to M$, such that $(\gamma, \rho) \models_{\text{ML}} (\exists V)\varphi$. Then, there exists $\rho' : Var \to M$, with $\rho'(Y) = \rho(Y)$, for all $Y \notin V$ such that $(\gamma, \rho') \models_{\text{ML}} \varphi$ (cf. Definition 2.3.3). Therefore, there exists a valuation ρ' such that $\gamma \in [\![\varphi]\!]$.

($\supseteq$) Let $\gamma \in [\![\varphi]\!]$. By Definition 2.3.5, there exists $\rho : Var \to M$, such that $(\gamma, \rho) \models_{\text{ML}} \varphi$. We have to show that there is a valuation $\rho' : Var \to M$ such that $(\gamma, \rho') \models_{\text{ML}} (\exists V)\varphi$. Let $\rho' = \rho$. Then, we have $\rho(Y) = \rho'(Y)$ for all $Y \notin V$ and $(\gamma, \rho) \models_{\text{ML}} \varphi$. Thus, cf. Definition 2.3.3, $(\gamma, \rho) \models_{\text{ML}} (\exists V)\varphi$ and $\gamma \in [\![(\exists V)\varphi]\!]$.

■

The set of free variables occurring in an ML formula $\varphi \triangleq \pi \wedge \phi$ is $vars(\varphi) \triangleq vars(\pi) \cup vars(\phi)$, where $vars(\pi)$ and $vars(\phi)$ are defined in Section 2.1 and Section 2.2, respectively. For some of our subsequent results we are also interested well-definedness of patterns:

Definition 2.3.6 ((Weakly) well-defined patterns) *Let φ be a pattern. Then φ is* weakly well-defined *iff for any valuation $\rho : Var \to \mathcal{T}$ there is some configuration $\gamma \in \mathcal{T}_{Cfg}$ such that $(\gamma, \rho) \models_{\text{ML}} \varphi$, and φ is* well-defined *if γ is unique.*

Example 2.3.4 *Let $\varphi \triangleq \langle\langle \texttt{if}\ B\ \{\ S\ \}\rangle\langle E\rangle\rangle \wedge (B = true)$. Then, the pattern φ is weakly well-defined because there are a configuration $\gamma \triangleq \langle\langle \texttt{if}\ true\ \{\ \texttt{skip};\ \}\rangle\langle \texttt{x} \mapsto 0\rangle\rangle$ and a valuation $\rho : Var \to M$, where $\rho(B) = true$, $\rho(S) = \texttt{skip};$, and $\rho(E) = \texttt{x} \mapsto 0$, such that $(\gamma, \rho) \models_{\text{ML}} \varphi$. The pattern $\varphi' \triangleq \langle\langle \texttt{if}\ B\ \{\ \texttt{skip};\ \}\rangle\langle \texttt{x} \mapsto 0\rangle\rangle \wedge (B = true)$ is well defined, since there is an unique configuration $\gamma' = \gamma$ such that $(\gamma', \rho) \models_{\text{ML}} \varphi'$. On the other hand, the pattern $\varphi'' \triangleq \langle\langle \texttt{if}\ B\ \{\ S\ \}\rangle\langle E\rangle\rangle \wedge false$ is not weakly well-defined because it is impossible to find a configuration γ'' and valuation ρ'' such that $(\gamma'', \rho'') \models_{\text{ML}} \varphi''$.*

2.4 Reachability Logic

Reachability Logic [98, 101, 100, 93] is a logic designed to use operational semantics for program verification and reasoning about programs. It includes a *language-independent* proof system for deriving reachability properties, which takes any operational semantics of a programming language, and derives any program properties that can be derived with *language-specific* proof systems (e.g. Hoare logics or dynamic logics). The main ingredients of RL are *reachability rules*, also called RL formulas, which generalise both term-rewrite rules and Hoare triples. The proof system derives reachability rules using the trusted operational semantics of the target language. Thus, the verification of a program reduces to checking if its specification (given as a reachability rule) is derivable.

In contrast to ML, which is a static logic of program configurations designed to state and reason about their structural properties, RL can be seen as a dynamic logic of configurations, expressing their evolution over time. In Section 2.3 we showed how one ML pattern φ specifies all the configurations γ that match against it. Here, specifications (i.e. reachability rules) are pairs of patterns, written $\varphi \Rightarrow \varphi'$.

In the following, we let $\mathcal{S}$ be a set of reachability rules and an underlying ML signature $(\Sigma, \Pi, \mathit{Cfg})$. $\mathcal{S}$ induces a transition system on any Σ-algebra/model M. Let us fix an arbitrary model M, which we call *configuration model*, and a sort-wise set of variables Var. We now formally define reachability rules an the transition system induced by $\mathcal{S}$.

Definition 2.4.1 (RL formula) *A* reachability rule *or a* RL formula *is an expression of the form $\varphi \Rightarrow \varphi'$ where φ and φ' are ML formulas. A* reachability system *is a set of reachability rules.*

A reachability rule $\varphi \Rightarrow \varphi'$ specifies that configurations that match against φ eventually transit to configurations that match against φ'. In other words, the transitions over program configurations are instances of reachability rules.

Example 2.4.1 *Let us consider the program configuration $\langle\langle \texttt{code} \rangle \langle m \rangle\rangle$ shown in Example 2.3.1. Then, $\langle\langle \texttt{x} = I; \texttt{skip} \rangle \langle \texttt{x} \mapsto J \rangle\rangle \Rightarrow \langle\langle \texttt{skip} \rangle \langle \texttt{x} \mapsto I \rangle\rangle$ is a reachability rule specifying that configurations that match against the*

pattern $\langle\langle \mathtt{x} = I; \mathtt{skip}\rangle\langle \mathtt{x} \mapsto J\rangle\rangle$*, (e.g.* $\langle\langle \mathtt{x} = 5; \mathtt{skip}\rangle\langle \mathtt{x} \mapsto 10\rangle\rangle$*) transit to configurations that match against* $\langle\langle \mathtt{skip}\rangle\langle \mathtt{x} \mapsto I\rangle\rangle$ *(e.g.* $\langle\langle \mathtt{skip}\rangle\langle \mathtt{x} \mapsto 5\rangle\rangle$*).*

Any reachability system $\mathcal{S}$ induces a transition system on the configuration model:

Definition 2.4.2 (RL transition system) *$\mathcal{S}$ induces a transition system $(M_{Cfg}, \Rightarrow_{\mathcal{S}})$, where $\Rightarrow_{\mathcal{S}} \subseteq M_{Cfg} \times M_{Cfg}$ is defined by $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ iff there is $\varphi \Rightarrow \varphi'$ in $\mathcal{S}$ and $\rho : Var \to M$ with $(\gamma, \rho) \models_{\text{ML}} \varphi$ and $(\gamma', \rho) \models_{\text{ML}} \varphi'$.*

The transition system $(M_{Cfg}, \Rightarrow_{\mathcal{S}})$ contains the set of all transitions $\gamma \Rightarrow_{\mathcal{S}} \gamma'$, and thus, it captures precisely how the language defined by $\mathcal{S}$ operates. We extend the notation $[\![\_]\!]$ to rules, $[\![\alpha \triangleq \varphi \Rightarrow \varphi']\!] = \{\gamma \Rightarrow_{\{\alpha\}} \gamma' \mid (\exists \rho : Var \to M)((\gamma, \rho) \models_{\text{ML}} \varphi \wedge (\gamma', \rho) \models_{\text{ML}} \varphi')\}$. The next lemma shows that a RL formula of the form $(\exists X)\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi'$ generates the same transitions as $\pi \wedge \phi \Rightarrow \pi' \wedge \phi'$.

Lemma 2.4.1 *If $X \cap Y = \emptyset$ then $[\![(\exists X)\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi']\!] = [\![\pi \wedge \phi \Rightarrow \pi' \wedge \phi']\!]$.*

Proof We may assume without loss of generality that $X \cap vars(\pi', \phi') = \emptyset$ and $Y \cap vars(\pi, \phi) = \emptyset$. Note that the condition $X \cap Y = \emptyset$ from the hypothesis of the lemma, can be easily obtained by variable renaming.

($\subseteq$) Assume $\gamma \Rightarrow_{\mathcal{S}} \gamma' \in [\![(\exists X)\pi \wedge \phi \Rightarrow (\exists Y)\pi' \wedge \phi']\!]$. Then, there is ρ such that $(\gamma, \rho) \models_{\text{ML}} (\exists X)\pi \wedge \phi$ and $(\gamma', \rho) \models_{\text{ML}} (\exists Y)\pi' \wedge \phi'$. By Definition 2.3.3, there are ρ_X and ρ_Y such that $(\gamma, \rho_X) \models_{\text{ML}} \pi \wedge \phi$ and $\rho_X(z) = \rho(z)$ for all $z \in Var \setminus X$, and $(\gamma', \rho_Y) \models_{\text{ML}} \pi' \wedge \phi'$ and $\rho_Y(z) = \rho(z)$ for all $z \in Var \setminus Y$. Let $\rho' = \rho_X \uplus \rho_Y$. Note that ρ' is well-defined since ρ_X and ρ_Y coincide on $Var \setminus (X \cup Y)$. Obviously, we have $(\gamma, \rho') \models_{\text{ML}} \pi \wedge \phi$ (ρ' and ρ_X coincide on $vars(\pi, \phi)$) and $(\gamma', \rho') \models_{\text{ML}} \pi' \wedge \phi'$ (ρ' and ρ_Y coincide on $vars(\pi', \phi')$). So, $\gamma \Rightarrow_{\mathcal{S}} \gamma' \in [\![\pi \wedge \phi \Rightarrow \pi' \wedge \phi']\!]$.

($\supseteq$) Assume $\gamma \Rightarrow_{\mathcal{S}} \gamma' \in [\![\pi \wedge \phi \Rightarrow \pi' \wedge \phi']\!]$. There is ρ such that $(\gamma, \rho) \models_{\text{ML}} \pi \wedge \phi$ and $(\gamma', \rho) \models_{\text{ML}} \pi' \wedge \phi'$. Indeed, we have $(\gamma, \rho) \models_{\text{ML}} (\exists X)\pi \wedge \phi$ and $(\gamma', \rho) \models_{\text{ML}} (\exists Y)\pi' \wedge \phi'$.

■

Lemma 2.4.1 allows us to omit explicitly the writing of the existential quantifiers in RL formulas.

In the following lemma we show that the transition system does not change if, in a rule $\pi \wedge \phi \Rightarrow \pi' \wedge \phi'$, we replace a subterm in π with a variable, and we add the equality between the variable and the replaced term to ϕ. If $\alpha \triangleq \varphi \Rightarrow \varphi'$, we denote by $vars(\alpha)$ the set $vars(\varphi) \cup vars(\varphi')$.

Lemma 2.4.2 *Let $\alpha \triangleq \pi \wedge \phi \Rightarrow \pi' \wedge \phi'$ be a rule in $\mathcal{S}$, p a position in π, and a rule $\alpha' \triangleq \pi[X]_p \wedge (\phi \wedge X = \pi|_p) \Rightarrow \pi' \wedge \phi'$, where $X \notin vars(\alpha)$ is a fresh variable which has the same sort as $\pi|_p$. Then $[\![\alpha]\!] = [\![\alpha']\!]$.*

Proof First, we prove that for all valuations $\rho : Var \to M$ and configurations γ, such that $(\gamma, \rho) \models_{\text{ML}} \pi[X]_p \wedge (\phi \wedge X = \pi|_p)$ we have $\rho(X) = \pi|_p$ (♠). Proof by contradiction: let ρ be a valuation and γ a configuration such that $(\gamma, \rho) \models_{\text{ML}} \pi[X]_p \wedge (\phi \wedge X = \pi|_p)$ and $\rho(X) \neq \pi|_p$. From $(\gamma, \rho) \models_{\text{ML}} \pi[X]_p \wedge (\phi \wedge X = \pi|_p)$ we obtain $\rho \models_{\text{ML}} (\phi \wedge X = \pi|_p)$. Since $X \notin vars(\alpha)$ (i.e. $X \notin vars(\phi)$) then $\rho \models_{\text{ML}} \phi$ and $\rho \models_{\text{ML}} X = \pi|_p$. But $X \notin vars(\pi)$, so the only possible valuation of X is $\rho(X) = \pi|_p$, which is a contradiction.

($\subseteq$) Let $\gamma \Rightarrow \gamma' \in [\![\alpha]\!]$. Then, there exists $\rho : Var \to M$ such that $(\gamma, \rho) \models_{\text{ML}} \pi \wedge \phi$ and $(\gamma', \rho) \models_{\text{ML}} \pi' \wedge \phi'$. Let $\rho' : Var \to M$ be a valuation defined as $\rho'(Y) = \rho(Y)$ for $Y \in Var \setminus \{X\}$, and $\rho'(X) = \pi|_p$. Since $X \notin vars(\alpha)$ and $(\gamma', \rho) \models \pi' \wedge \phi'$ we obtain $(\gamma', \rho') \models \pi' \wedge \phi'$. Moreover, we have $\rho'(\pi[X]_p) = \rho'(\pi[\pi|_p]_p) = \rho'(\pi) = \rho(\pi) = \gamma$, and $\rho'(\phi \wedge X = \pi|_p) = \rho'(\phi \wedge \pi|_p = \pi|_p) = \rho'(\phi) = \rho(\phi) = \top$. Therefore, there exists $\rho' : Var \to M$ such that $(\gamma, \rho') \models_{\text{ML}} \pi[X]_p \wedge (\phi \wedge X = \pi|_p)$ and $(\gamma', \rho) \models_{\text{ML}} \pi' \wedge \phi'$, i.e. $\gamma \Rightarrow \gamma' \in [\![\alpha']\!]$.

($\supseteq$) Let $\gamma \Rightarrow \gamma' \in [\![\alpha']\!]$. Then, there exists $\rho' : Var \to M$ such that $(\gamma, \rho') \models_{\text{ML}} \pi[X]_p \wedge (\phi \wedge X = \pi|_p)$ and $(\gamma, \rho') \models_{\text{ML}} \pi' \wedge \phi'$. Let $\rho : Var \to M$ be a valuation such that $\rho(Y) = \rho'(Y)$ for $Y \in Var \setminus \{X\}$. Then $\gamma = \rho'(\pi[X]_p) = \rho'(\pi[\pi|_p]_p) = \rho'(\pi) = \rho(\pi)$ (we used (♠) and the fact that $X \notin vars(\pi)$) and $\top = \rho'(\phi \wedge X = \pi|_p) = \rho'(\phi) = \rho(\phi)$ (we used (♠) and the fact that $X \notin vars(\pi, \phi)$). Therefore, there exists $\rho : Var \to M$, such that $(\gamma, \rho) \models_{\text{ML}} \pi \wedge \phi$ and $(\gamma', \rho) \models_{\text{ML}} \pi' \wedge \phi'$, i.e. $\gamma \Rightarrow \gamma' \in [\![\alpha]\!]$.
■

The following notions are used in the definition of the satisfaction relation over transition systems and RL formulas.

Definition 2.4.3 *A configuration* $\gamma \in M_{Cfg}$ terminates *in* $(M_{Cfg}, \Rightarrow_{\mathcal{S}})$ *iff there is no infinite sequence* $\gamma \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \gamma_2 \Rightarrow_{\mathcal{S}} \cdots$.

Definition 2.4.4 (Weakly well-defined) *A reachability rule* $\varphi \Rightarrow \varphi'$ *is* weakly well-defined*, respectively* well-defined*, if* φ' *is weakly well-defined, respectively well-defined (cf. Definition 2.3.6).*

RL formulas are used to express program properties. Semantic validity in RL follows the same line of partial correctness in Hoare Logic, but in more general terms of reachability:

Definition 2.4.5 (RL semantics) *The satisfaction relation* $\models_{\text{RL}}$ *between the transition systems* $(M_{Cfg}, \Rightarrow_{\mathcal{S}})$ *and RL formulas* $\varphi \Rightarrow \varphi'$ *is defined as follows:* $(M_{Cfg}, \Rightarrow_{\mathcal{S}}) \models_{\text{RL}} \varphi \Rightarrow \varphi'$ *iff for all* $\gamma \in M_{Cfg}$ *such that* γ *terminates in* $(M_{Cfg}, \Rightarrow_{\mathcal{S}})$ *and for all* $\rho : Var \to M$ *such that* $(\gamma, \rho) \models_{\text{ML}} \varphi$*, there exists some* $\gamma' \in M_{Cfg}$ *such that* $(\gamma', \rho) \models_{\text{ML}} \varphi'$ *and* $\gamma \Rightarrow_{\mathcal{S}}^{*} \gamma'$.

By $\Rightarrow_{\mathcal{S}}^{*}$ we denote the transitive closure of the relation $\Rightarrow_{\mathcal{S}}$. Intuitively, $(M_{Cfg}, \Rightarrow_{\mathcal{S}}) \models_{\text{RL}} \varphi \Rightarrow \varphi'$ specifies reachability: any configuration γ that terminates and is matched against φ transits, on some execution path, to a configuration γ' that is matched against φ'.

Since the model M and the reachability system S are fixed, we often write $\mathcal{S} \models_{\text{RL}} \varphi \Rightarrow \varphi'$ instead of $(M_{Cfg}, \Rightarrow_{\mathcal{S}}) \models_{\text{RL}} \varphi \Rightarrow \varphi'$. We let $\models_{\text{FOL}} \varphi \to \varphi'$ denote the fact that the FOL implication between the FOL encoding of ML patterns φ and φ' is valid (see [98] for details).

The proof system of RL is shown in Figure 2.1. It derives sequents of the form $\mathcal{S} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'$, where $\mathcal{S}$ and $\mathcal{C}$ are sets of RL formulas. The rules in $\mathcal{S}$ are called *axioms*, while the ones in $\mathcal{C}$ are called *circularities*. When $\mathcal{C}$ is empty ($\emptyset$) we write $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ instead of $\mathcal{S} \vdash_{\emptyset} \varphi \Rightarrow \varphi'$. During the proof, the set $\mathcal{C}$ is populated with circularities via [Circularity].The [Axiom] proof rule states that the rules in $\mathcal{S}$ can be used as initial axioms. Both [Reflexivity] and [Transitivity] stand for the corresponding properties of the reachability relation, while [Consequence] and [CaseAnalysis] are adapted from Hoare Logic. [Abstraction] allows hiding irrelevant details of φ behind an existential quantifier. This deduction rule is particularly useful in combination with [Circularity] which has a co-inductive nature. [Circularity] is used, e.g., to prove properties about loops, recursive functions, jumps,

$$\text{[Axiom]}\ \frac{\varphi \Rightarrow \varphi' \in \mathcal{S} \quad \phi \text{ is a (patternless) FOL formula}}{\mathcal{S} \vdash_{\mathcal{C}} \varphi \wedge \phi \Rightarrow \varphi' \wedge \phi}$$

$$\text{[Abstraction]}\ \frac{\mathcal{S} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi' \quad X \cap vars(\varphi') = \emptyset}{\mathcal{S} \vdash_{\mathcal{C}} ((\exists X)\, \varphi \Rightarrow \varphi')}$$

$$\text{[Reflexivity]}\ \frac{\cdot}{\mathcal{S} \vdash \varphi \Rightarrow \varphi}$$

$$\text{[Consequence]}\ \frac{\models_{\text{FOL}} \varphi_1 \rightarrow \varphi_1' \quad \mathcal{S} \vdash_{\mathcal{C}} \varphi_1' \Rightarrow \varphi_2' \quad \models_{\text{FOL}} \varphi_2' \rightarrow \varphi_2}{\mathcal{S} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi_2}$$

$$\text{[CaseAnalysis]}\ \frac{\mathcal{S} \vdash_{\mathcal{C}} \varphi_1 \Rightarrow \varphi \quad \mathcal{S} \vdash_{\mathcal{C}} \varphi_2 \Rightarrow \varphi}{\mathcal{S} \vdash_{\mathcal{C}} (\varphi_1 \vee \varphi_2) \Rightarrow \varphi}$$

$$\text{[Transitivity]}\ \frac{\mathcal{S} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'' \quad (\mathcal{S} \cup \mathcal{C}) \vdash \varphi'' \Rightarrow \varphi'}{\mathcal{S} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

$$\text{[Circularity]}\ \frac{\mathcal{S} \vdash_{\mathcal{C} \cup \{\varphi \Rightarrow \varphi'\}} \varphi \Rightarrow \varphi'}{\mathcal{S} \vdash_{\mathcal{C}} \varphi \Rightarrow \varphi'}$$

Figure 2.1: The language-independent proof system of Reachability Logic.

etc. In particular, it allows us to make a claim of circular behaviour at any moment during a proof derivation; the claim holds if it can be proved using itself. This type of reasoning is sound because the circularity claim will be allowed only after at least one application of a (trusted) rule from $\mathcal{S}$. In addition to the proof rules from Figure 2.1, the proof system of RL also contains the following derived rules (proved in [93]):

- *Substitution*: $\mathcal{S} \vdash_G \theta(\varphi) \Rightarrow \theta(\varphi')$, if $\theta : Var \rightarrow T_\Sigma(Var)$ and $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$;
- *Logical Framing*: $\mathcal{S} \vdash_G (\varphi \wedge \phi) \Rightarrow (\varphi' \wedge \phi)$, if ϕ is a patternless FOL formula and $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$;
- *Set Circularity*: if $\mathcal{S} \vdash_G \varphi \Rightarrow \varphi'$ for each $\varphi \Rightarrow \varphi' \in G$ and G is finite then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ for each $\varphi \Rightarrow \varphi' \in G$;
- *Implication*: if $\models_{\text{FOL}} \varphi \rightarrow \varphi'$ then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$;
- *Monotony*: if $\mathcal{S} \subseteq \mathcal{S}'$ then $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ implies $\mathcal{S}' \vdash \varphi \Rightarrow \varphi'$.

In this dissertation we will make use of the following theorem, which states that the proof system shown in Figure 2.1 is sound:

Theorem 2.4.1 (Soundness) *If $\mathcal{S}$ is weakly well-defined and $\mathcal{S} \vdash \varphi \Rightarrow \varphi'$ then $\mathcal{S} \models_{\text{RL}} \varphi \Rightarrow \varphi'$.*

The proof of Theorem 2.4.1 can be found in [93].

2.5 The $\mathbb{K}$ framework

In this section we present the basics of $\mathbb{K}$, a framework for defining formal semantics of programming languages. $\mathbb{K}$ has been used in this dissertation as a platform on top of which we have implemented our symbolic execution framework. This brought us a few advantages in terms of implementation due to the fact that our symbolic execution tool inherited all the features of $\mathbb{K}$. Moreover, we took advantage of the existing language definitions by testing our symbolic execution on the defined languages. We describe the main features of the $\mathbb{K}$ framework by means of an existing language definition called CinK, which is presented entirely in [70]. CinK is an overly simplified core of the C++ language, which includes the basic integer and boolean expressions, local and global variables, side effects, lvalues and rvalues, functions, call by value, (unidimensional) arrays, pointers, and the basic imperative statements (conditionals, loops, etc.). In [71], the language has been also extended with a few concurrency constructs (e.g. thread creation, lock-based synchronisation, thread-join), but here we stick to the "non-concurrent" version of CinK, since we only highlight the basic ingredients of a $\mathbb{K}$ definition.

Introduced by Grigore Roşu in 2003 [91] for teaching a programming languages class, and continuously refined and developed ever since, $\mathbb{K}$ [97, 108] is a framework for defining programming language based on rewriting which combines the strengths of existing frameworks (expressiveness, modularity, concurrency, and simplicity) while avoiding their weaknesses. If one ignores its concurrent semantics, $\mathbb{K}$ can be seen as a notation within rewriting logic [79], the same as most other semantic frameworks, such as natural (or big-step) semantics, (small-step) SOS, Modular SOS, reduction semantics with evaluation contexts, and so on [111]. However, unlike these other semantic frameworks enumerated above, $\mathbb{K}$ cannot be easily

captured step-for-step in rewriting logic, due to its enhanced concurrency which is best described in terms of ideas from graph rewriting [110].

$\mathbb{K}$ has been successfully used to define a large number of programming languages, from simple languages used for teaching (e.g. CinK, SIMPLE [102], KOOL [53], KernelC [113]) to real-life programming languages, such as, C [35], Java [17], Scheme [85], Python [51], PHP [39]. Some other $\mathbb{K}$ definitions were developed to capture various aspects of features of OCL [6], RISC assembly [14, 11], Verilog [76], Javascript [80], Haskell'98 [66], X10 [44], and LLVM [68]. The framework was also used for designing type-checkers [36], model-checking with predicate abstractions [9, 13, 8], path directed symbolic execution [10], program equivalence [69], computing worst-case time analysis [11, 12], and runtime verification [96]. The theoretical foundations of the $\mathbb{K}$ framework can be found in [97], while the $\mathbb{K}$ compiler and runner are described in [114] and [67], respectively.

The main ingredients of a $\mathbb{K}$ semantic definition of a language are *computations*, *configurations* and *rules*. Computations are special tasks which can be seen as *units* of program execution. Configurations are nested structures which hold both syntactic and semantics information about programs. The $\mathbb{K}$ rules are a particular type of reachability rules, where modularity is achieved by specifying only those parts that change in a configuration. All these ingredients are illustrated on the $\mathbb{K}$ definition of CinK.

The *syntax* of a language is given using a BNF-style grammar. Here we only exhibit a fragment of the $\mathbb{K}$ definition of CinK (Figure 2.2). The grammar productions are usually annotated with $\mathbb{K}$-specific attributes. Some of them are related to the evaluation order of the arguments of a specific syntactical construct. For instance, a major feature of the C++ expressions is the "sequenced before" relation [57], which defines a partial order over the evaluation of subexpressions. This can be easily expressed in $\mathbb{K}$ using the *strict* attribute to specify an evaluation order for an operation's operands. If the operator is annotated with the *strict* attribute then its operands will be evaluated in a nondeterministic order. For instance, all the binary operations are strict. Hence, they may induce non-determinism in programs because of possible side-effects in their arguments.

Another feature is given by the classification of expressions into *rval-*

Exp	::=	*Id* | *Int* | *Bool* | *String*	
	|	*Exp* `(` *Exp* `)`	$[strict(1(context(rvalue))), funcall]$
	|	`++` *Exp*	$[strict, prefinc]$
	|	`--` *Exp*	$[strict, prefdec]$
	|	*Exp* `/` *Exp*	$[strict(all(context(rvalue))), divide]$
	|	*Exp* `+` *Exp*	$[strict(all(context(rvalue))), plus]$
	|	*Exp* `>` *Exp*	$[strict(all(context(rvalue)))]$
	|	*Exp* `&&` *Exp*	$[strict, and]$
Stmt	::=	*Exps* `;`	$[strict]$
	|	`{`*Stmts*`}`	
	|	`while` `(`*Exp*`)`*Stmt*	
	|	`return` *Exp* `;`	$[strict(all(context(rvalue)))]$
	|	*Decl* `(` *Decls* `){` *Stmts* `}`	$[fundecl]$
	|	`if` `(`*Exp*`)`*Stmt* `else` *Stmt*	$[strict(1(context(rvalue)))]$
Decl	::=	*Type Exp*	
Type	::=	`int` | `bool` | `void`	

Figure 2.2: Fragment of CinK syntax

ues and *lvalues*. The arguments of binary operations are evaluated as rvalues and their results are also rvalues, while, e.g., both the argument of the prefix-increment operation and its result are lvalues. The *strict* attribute for such operations has a sub-attribute *context* for wrapping any subexpression that must be evaluated as an rvalue. Other attributes ($funcall, divide, plus, minus, \dots$) are names associated to each syntactic production, which can be used to refer to them.

A $\mathbb{K}$ *configuration* is used to store program states. It consists in a nested structure of cells, which typically includes the program to be executed, input and output streams, values for program variables, and other additional information. The configuration of CinK (Figure 2.3) includes the $\langle\rangle_{\mathsf{k}}$ cell containing the code that remains to be executed, which is represented as a list of *computation tasks* $C_1 \curvearrowright C_2 \curvearrowright \dots$ to be executed in the given order. Computation tasks are typically statements and expression evaluations. The memory is modelled using two cells $\langle\rangle_{\mathsf{env}}$ (which

$$\langle\ \langle \$PGM \rangle_{\mathsf{k}}\ \ \langle\cdot\rangle_{\mathsf{env}}\ \ \langle\cdot\rangle_{\mathsf{store}}\ \ \langle\cdot\rangle_{\mathsf{stack}}\ \ \langle\cdot\rangle_{\mathsf{return}}\ \ \langle\cdot\rangle_{\mathsf{in}}\ \ \langle\cdot\rangle_{\mathsf{out}}\ \ \rangle_{\mathsf{cfg}}$$

Figure 2.3: CinK configuration

holds a map from variables to memory locations) and $\langle\rangle_{\mathsf{state}}$ (which holds a map from memory locations to values). Locations are used as values for pointer types. The configuration also includes a cell for the function call stack $\langle\rangle_{\mathsf{stack}}$ and another one $\langle\rangle_{\mathsf{return}}$ for the return values of functions. The $\langle\rangle_{\mathsf{in}}$ and $\langle\rangle_{\mathsf{out}}$ cells hold the input and (respectively) the output of the program. Both cells are connected to the standard input/output stream. When the configuration is initialised at runtime, a CinK program is loaded in the $\langle\rangle_{\mathsf{k}}$ cell, and all the other cells remain empty.

A $\mathbb{K}$ *rule* is a topmost rewrite rule specifying transitions between configurations. Since usually only a small part of the configuration is changed by a rule, a *configuration abstraction* mechanism is used, allowing one to only specify the parts transformed by the rule. For instance, the (abstract) rule for addition, shown in Figure 2.4, represents the (concrete) rule

$$\langle\langle I_1 \texttt{ + } I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\langle I\rangle_{\mathsf{in}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$
$$\Rightarrow$$
$$\langle\langle I_1 +_{Int} I_2 \curvearrowright C\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{env}}\langle S\rangle_{\mathsf{store}}\langle T\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}\langle I\rangle_{\mathsf{in}}\langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

where $+_{Int}$ is the mathematical operation for addition, I_1, I_2, V are integers, E and S are maps, and T, I, and O are lists. The three dots "$\cdots$" in a cell are meant to help the configuration abstraction mechanism, by specifying that there is some other content in the cell, which is not affected by the current rule.

The rule for division has a side condition which restricts its application, i.e. it applies only when the denominator is not zero. The conditional statement `if` has two corresponding rules, one for each possible evaluation of the condition expression. The rule for the `while` loop performs an unrolling into an `if` statement. The increment and update rules have side effects in the $\langle\rangle_{\mathsf{store}}$ cell, modifying the value stored at a specific address. Reading a value from the memory is specified by the lookup rule, which matches a value in the $\langle\rangle_{\mathsf{store}}$ and places it in the $\langle\rangle_{\mathsf{k}}$ cell. The auxiliary construct `$lookup` is used when a program variable is evaluated as an rvalue. The semantics of a variable declaration consists in adding new entries in the $\langle\rangle_{\mathsf{env}}$ and $\langle\rangle_{\mathsf{store}}$ cells, where the variable is mapped to a *fresh* memory location L, which is initialised with zero. A function declaration is handled in the same way as a variable declaration, except that the value stored in the fresh location is the lambda abstraction of the function. The rule defining the evaluation of a function is a bit more complex: it evaluates

I_1 `/` $I_2 \wedge I_2 \neq_{Int} 0 \Rightarrow I_1 /_{Int} I_2$ [*division*]
I_1 `+` $I_2 \Rightarrow I_1 +_{Int} I_2$ [*plus*]
I_1 `>` $I_2 \Rightarrow I_1 >_{Int} I_2$ [*greater*]

`if(` *true* `)` St `else` `_` $\Rightarrow St$ [*if-true*]
`if(` *false* `)` `_` `else` $St \Rightarrow St$ [*if-false*]

`while(` B `)` $St \Rightarrow$ `if(` B `){` St `while(` B `)` St `else` `{}}` [*while*]
V `;` $\Rightarrow \cdot$ [*stmt-exp*]

$\langle$`++lvalue(` L `)` $\Rightarrow$ `lvalue(` L `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto (V \Rightarrow V +_{Int} 1) \cdots\rangle_{\mathsf{store}}$ [*inc*]
$\langle$`--lvalue(` L `)` $\Rightarrow$ `lvalue(` L `)` $\cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto (V \Rightarrow V -_{Int} 1) \cdots\rangle_{\mathsf{store}}$ [*dec*]
$\langle\langle$`lvalue(` L `)=` $V \Rightarrow V \cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto \_ \Rightarrow V \cdots\rangle_{\mathsf{store}} \cdots\rangle_{\mathsf{cfg}}$ [*update*]
$\langle\langle$`$lookup(` L `)` $\Rightarrow V \cdots\rangle_{\mathsf{k}}\langle\cdots L \mapsto V \cdots\rangle_{\mathsf{store}} \cdots\rangle_{\mathsf{cfg}}$ [*lookup*]

`{` *Sts* `}` $\Rightarrow$ *Sts* [*block*]

$\langle T X$ `;`$\Rightarrow \cdot \cdots\rangle_{\mathsf{k}}\langle\cdots Env \Rightarrow Env[L/X] \cdots\rangle_{\mathsf{env}}$
$\langle\cdots \cdot \Rightarrow L \mapsto 0 \cdots\rangle_{\mathsf{store}} \wedge fresh(L)$ [*vardecl*]

$\langle D$: *Decl* `(` X : *Decls* `){` S : *Stmts* `}`$\Rightarrow \cdot \cdots\rangle_{\mathsf{k}}$
$\langle\cdots \cdot \Rightarrow$`getName(` D `)`$\mapsto L \cdots\rangle_{\mathsf{env}}$
$\langle\cdots \cdot \Rightarrow L \mapsto \lambda X.S \cdots\rangle_{\mathsf{store}} \wedge fresh(L)$ [*fundecl*]

$\langle\lambda X.S(E) \Rightarrow$`eval app(`$E$`)to` $\cdot_{List}$ `fol` X`;`$\curvearrowright$`seqpoint`$\curvearrowright \lambda X.S(\square) \cdots\rangle_{\mathsf{k}}$ [*funcall*]

$\langle$`eval app(`$\cdot_{Exps}$`)to` V `fol` X`;`$\curvearrowright$`seqpoint`$\Rightarrow$
`seqpoint`$\curvearrowright$`eval app(`$\cdot_{Exps}$`)to` V `fol` X`;`$\cdots\rangle_{\mathsf{k}}$
$\langle$`eval app(`$\cdot_{Exps}$`)to` V `fol` Y`;`$\curvearrowright \lambda X.S(\square) \curvearrowright K\rangle_{\mathsf{k}}\langle E\rangle_{\mathsf{stack}} \Rightarrow$
`bind` V `to` X`;`$\curvearrowright S \curvearrowright$`return noVal;`$\cdots\rangle_{\mathsf{k}}\langle([Env], K)E\rangle_{\mathsf{stack}}$

$\langle$`bind(`V, Vs`) to (` T X, Xl `);`$\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\langle M\rangle_{\mathsf{store}} \Rightarrow$
$\langle$`bind(`Vs`) to (` Xl `);`$\rangle_{\mathsf{k}}\langle Env[L/X]\rangle_{\mathsf{env}}\langle M\ L \mapsto V\rangle_{\mathsf{store}}$
$\langle$`bind(`$\cdot_{Exps}$`) to (` $\cdot_{Decls}$ `);`$\rangle_{\mathsf{k}} \Rightarrow \langle\cdot_K\rangle_{\mathsf{k}}$

$\langle$`return(`V`);`$\curvearrowright$ `_`$\rangle_{\mathsf{k}}\langle$`_`$\rangle_{\mathsf{env}}\langle([Env], K)\rangle_{\mathsf{stack}}\langle$`_`$\rangle_{\mathsf{return}} \Rightarrow$
$\langle V \curvearrowright K\rangle_{\mathsf{k}}\langle Env\rangle_{\mathsf{env}}\langle\cdot_{List}\rangle_{\mathsf{stack}}\langle V\rangle_{\mathsf{return}}$ [*return*]

Figure 2.4: Subset of rules from the $\mathbb{K}$ semantics of CinK

the parameters, then binds values to the formal parameters, and then it executes the body while saving the calling context. We use $\square$ as a special variable, destined to receive the result of an evaluation.

In addition to these rules (written by the $\mathbb{K}$ user), the $\mathbb{K}$ framework automatically generates so-called *heating* and *cooling* rules, which are induced by *strict* attributes. We show the case of division, which is strict in both arguments:

$$A_1 \ / \ A_2 \Rightarrow \mathit{rvalue}(A_1) \curvearrowright \square \ / \ A_2 \qquad \mathit{rvalue}(I_1) \curvearrowright \square \ / \ A_2 \Rightarrow I_1 \ / \ A_2$$
$$A_1 \ / \ A_2 \Rightarrow \mathit{rvalue}(A_2) \curvearrowright A_1 \ / \ \square \qquad \mathit{rvalue}(I_2) \curvearrowright A_1 \ / \ \square \Rightarrow A_1 \ / \ I_2$$

The $\mathbb{K}$ rules are classified in two categories: *structural* rules and *computational* rules. Intuitively, only the computational rules yield transitions in the transition system associated to a program, while the structural rules are meant to only re-arrange the configuration so that the computational rules can match.

In terms of implementation, $\mathbb{K}$ consists in a set of command line tools which includes a definition compiler and a program runner. The $\mathbb{K}$ compiler generates a rewrite theory, which is then used to run real programs. At runtime, the runner first loads the given program in the initial configuration and then applies repeatedly the $\mathbb{K}$ rules to the configuration, until no rule can be applied. The result consists in one or more *final* configurations. $\mathbb{K}$ uses an external rewrite engine (Maude), which allows I/O interactions [7], and is integrated with the Z3 SMT solver [34]. Thus, $\mathbb{K}$ can be used as a tool for both executing the formal semantics of a programming language and for analysing programs written in that language.

Chapter 3

Language Independent Symbolic Execution Framework

Symbolic execution is a well-known program-analysis technique used in testing, verification, debugging, and other applications. There are many tools implementing symbolic execution engines but most of them are language dependent (e.g. Java PathFinder [82], PEX [38]). Any existent language independent approaches are based on translations to different languages (e.g. WOODPECKER [31], KLEE [23]), where symbolic execution is implemented at the compiler level. However, this approach has a few downsides mostly related to the fact that the translation of a language into another language is not guaranteed to be sound.

The main goal of this dissertation is to design a language independent framework which is based on the language's formal semantics. In this chapter, we introduce formal definitions for programming languages and symbolic execution. We prove that the symbolic execution thus defined has the following properties, which ensures a natural relation between symbolic and concrete program execution:

- *Coverage*: to every concrete execution there corresponds a feasible symbolic one;
- *Precision*: to every feasible symbolic execution there corresponds a concrete one;

Two executions are said to be corresponding if they take the same path in the control flow graph of the program, and a symbolic execution is feasible if the path conditions along it are satisfiable. Or, stated in terms of simulations: the feasible symbolic executions and the concrete executions of any given program mutually simulate each other. These properties are very important when performing analyses on symbolic programs, since the results of those analyses can be soundly transferred to concrete instances of the symbolic programs in question.

This chapter is organised as follows: we define programming languages in terms of algebraic specifications and RL formulas in Section 3.1. Then, we present (what we mean by) unification (Section 3.2) and we prove a technical lemma which states that unification can be resolved by matching under reasonable conditions. Unification is then used to define symbolic execution and the lemma is used to prove the *Coverage* and *Precision* properties in Section 3.3. Finally, in Section 3.4, we show how our symbolic execution framework can be implemented by standard rewriting.

3.1 Language Definitions

A wide range of programming languages are used nowadays for programming billions of hardware devices. Depending on their level of abstraction, there are low level programming languages, whose instructions are very close to the language circuits (e.g. Assembler, Verilog, VHDL), or high level imperative, functional or object-oriented programming languages (e.g. C, OCaml, Java, etc.). Whatever a programming language is, it requires a *specification* which clearly states the meaning of the programs written in that language. This specification includes the syntax of the language, that is, the constructs of which programs are made, and the semantics of the language, which describes the behaviour of each syntactical construct. Over the years, researchers reached a consensus on the idea that language specifications should be given using a formal specification language. However, formal specification languages have been themselves designed to be appropriate for a single analysis area: structural operational semantics for dynamic semantics, reduction semantics with evaluation contexts for proving type soundness, or axiomatic semantics for program verification. Moreover, formal descriptions of languages are

confined to the power of expression of their specification language.

In this dissertation we choose a formal specification language which is intended to keep the balance between simplicity and power of expressivity and analysis. We propose a general notion of language definition based on algebraic specifications and Reachability Logic. The syntax of a programming language is given using algebraic signatures, while its semantics is given using RL formulas. The intent of having this generic notion of language definition is driven by our goal of defining a generic framework for symbolic execution, which is parametric in the language definition.

We now define languages in terms of ML signatures and RL formulas.

Definition 3.1.1 (Language Definition) *A* language definition *is a tuple* $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$*, where:*

- $(\Sigma, \Pi, \mathit{Cfg})$ *is a ML signature,*
- $\mathcal{T}$ *is a model of* $(\Sigma, \Pi, \mathit{Cfg})$ *, and*
- $\mathcal{S}$ *is a set of RL formulas.*

In the rest of the dissertation, we consider that language definitions $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ meet the assumptions presented in the rest of this section.

Assumption 3.1.1 *The first-order signature* (Σ, Π) *includes a subsignature* $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ *consisting of all* data sorts *together with their operations and predicates. We implicitly consider a subset* $S^{\mathfrak{d}} \subset S$ *of data sorts and that* $\Sigma_{w,s} = \Sigma^{\mathfrak{d}}_{w,s}$ *for all* $(w, s) \in S^{\mathfrak{d}*} \times S^{\mathfrak{d}}$*. The signature* $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ *depends on the language* $\mathcal{L}$*, which has specific data sorts (e.g. integers, booleans, arrays, . . .). We always assume that the sort Cfg is not a data sort.*

The distinction between data and non-data sorts from Assumption 3.1.1 is essential, since in our approach we only allow the data sorts from $\Sigma^{\mathfrak{d}}$ to be symbolic. In Example 3.1.1 we show the signatures Σ and $\Sigma^{\mathfrak{d}}$, corresponding to the fragment of the syntax of CinK (Figure 2.2).

Example 3.1.1 *Consider the syntax fragment of CinK, shown in Figure 2.2. The set S includes nonterminals Int, Bool, String, Id, Exp, Exps,*

Stmt, and Stmts as sorts. The S-sorted signature Σ *is constructed as shown in Example 2.1.2, i.e. it includes the following S-indexed sets:* $\Sigma_{[],Int}$, $\Sigma_{[],Bool}$, $\Sigma_{[],String}$, $\Sigma_{[],Id}$, $\Sigma_{Int,Exp}$, $\Sigma_{Bool,Exp}$, $\Sigma_{Id,Exp}$, $\Sigma_{String,Exp}$, $\Sigma_{Exp,Exp}$, $\Sigma_{Exp\,Exp,Exp}$, $\Sigma_{Exp,Stmt}$, $\Sigma_{Exps,Stmt}$, $\Sigma_{Stmts,Stmt}$, $\Sigma_{Exp\,Stmt,Stmt}$, $\Sigma_{Exp\,Stmt\,Stmt,Stmt}$. *The data subsignature* $\Sigma^{\mathfrak{d}}$ *includes only the S-indexed sets* $\Sigma_{[],Bool}$, $\Sigma_{[],Int}$, $\Sigma_{[],Id}$, *and* $\Sigma_{[],String}$.

For CinK, the signature Σ also contains an operation symbol for configurations:

$$\langle\langle \_\rangle_{\mathsf{k}}\langle \_\rangle_{\mathsf{env}}\langle \_\rangle_{\mathsf{store}}\langle \_\rangle_{\mathsf{stack}}\langle \_\rangle_{\mathsf{return}}\langle \_\rangle_{\mathsf{in}}\langle \_\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \in \Sigma_{Stmts\,Map\,Map\,List\,K\,List\,List,Cfg},$$

where _ stands for the location of the arguments when using the infix notation.

Example 3.1.2 *The following term of sort Cfg:*

$$\langle\langle \mathtt{x} = \mathtt{y}; \mathtt{print(x)};\rangle_{\mathsf{k}}\langle \mathtt{x}\mapsto 7\ \mathtt{y}\mapsto 3\rangle_{\mathsf{env}}\langle 7\mapsto 5\ 3\mapsto 0\rangle_{\mathsf{store}}\langle\cdot\rangle_{\mathsf{stack}}\langle\cdot\rangle_{\mathsf{return}}\langle\cdot\rangle_{\mathsf{in}}\langle\cdot\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}}$$

is a valid configuration for CinK, where program variables x *and* y *are mapped to locations* 7 *and* 3 *respectively, which in* $\langle\rangle_{\mathsf{store}}$ *are mapped to values* 5 *and* 0*, respectively.*

Assumption 3.1.2 *We consider a fixed* $(\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$*-model* $\mathcal{D}$*. The elements of* $\mathcal{D}$ *are added to the signature* $(\Sigma, \Pi) \setminus (\Sigma^{\mathfrak{d}}, \Pi^{\mathfrak{d}})$ *as constants of their respective sorts. The model* $\mathcal{T}$ *is a* $\mathcal{D}$-grounded model *defined as in Definition 3.1.2.*

Definition 3.1.2 *A* $\mathcal{D}$-grounded model *is a* (Σ, Π)*-model* $\mathcal{T}$ *satisfying:*

1. *if* $c \in \Sigma^{\mathfrak{d}}$ *is a constant, then* $\mathcal{T}_c = \mathcal{D}_c$*;*
2. *if* $f \in \Sigma^{\mathfrak{d}}$ *is an operation symbol, then* $\mathcal{T}_f = \mathcal{D}_f$*;*
3. *if* $f, g \in \Sigma \setminus \Sigma^{\mathfrak{d}}$ *are two operation symbols with the same non-data result sort such that* $\mathcal{T}_f(\tau_1, \ldots \tau_n) = \mathcal{T}_g(\tau'_1, \ldots \tau'_m)$ *for some* $\tau_1, \ldots \tau_n$*,* $n \geq 0$ *and* $\tau'_1, \ldots \tau'_m$ *in* $\mathcal{T}$*,* $m \geq 0$*, then* $f = g$*,* $m = n$*, and* $\tau_i = \tau'_i$ *for all* $i \in \{1, \ldots, n\}$*.*
4. *if* $p \in \Pi^{\mathfrak{d}}$ *then* $\mathcal{T}_p = \mathcal{D}_p$*;*

Example 3.1.3 *For CinK, the model $\mathcal{D}$ interprets Int as the set of integers, the operations like* `+` *as the corresponding usual operation on integers, Bool as the set of Boolean values, the operations like* `&&` *as the usual Boolean operations, the sort Map as the set of mappings $X \mapsto I$, where X ranges over identifiers (Id) or integers (Int), and I over integers (Int).*

The model $\mathcal{T}$ interprets data terms in the same way as $\mathcal{D}$ (e.g. $\mathcal{T}_1 = \mathcal{D}_1$) and non-data terms as ground terms, where data subterms are replaced by their interpretations in $\mathcal{D}$. For instance, `if` $1 > 0$ `then { skip; } else { skip; }` *is interpreted as* $\mathcal{T}_{\texttt{if } \mathcal{D}_{true} \texttt{ then \{ skip; \} else \{ skip; \}}}$*, since $\mathcal{D}_{1>0} = \mathcal{D}_{true}$.*

Example 3.1.4 *Let size be a function symbol in $\Sigma_{Cfg,Int}$. Then the interpretation of size in $\mathcal{T}$ is given by $\mathcal{D}$, i.e. $\mathcal{T}_{size} = \mathcal{D}_{size}$.*

Assumption 3.1.3 *The last component of a language definition is a set $\mathcal{S}$ of RL formulas of the form $\pi \wedge \phi \Rightarrow \pi' \wedge \phi'$, where $vars(\phi) \subseteq vars(\pi)$. We tacitly assume that the variables in $vars(\pi' \wedge \phi')$ and not in $vars(\pi \wedge \phi)$ are existentially quantified.*

Example 3.1.5 *The $\mathbb{K}$ rules shown in Figure 2.4 are included in the set $\mathcal{S}$ of RL formulas which define the semantics of CinK. Note that $\mathbb{K}$ rules are in fact particular cases of RL formulas.*

3.2 Unification

Our approach uses unification for formally defining symbolic execution. In this section we define what we mean by unification and we prove a technical lemma which states that unification can be implemented using matching, under some assumptions. This result is essential for proving the *Coverage* and *Precision* properties, also formalised using unification.

We assume a given language definition $\mathcal{L} = ((\Sigma, \Pi, Cfg), \mathcal{T}, \mathcal{S})$ as in Section 3.1, and an infinite S-indexed set of variables Var.

Definition 3.2.1 (Unifiers) *A* symbolic unifier *of two terms t_1, t_2, with $vars(t_1) \cap vars(t_2) = \emptyset$, is any substitution $\sigma : vars(t_1) \cup vars(t_2) \to T_\Sigma(Z)$ for some set Z of variables such that $\sigma(t_1) = \sigma(t_2)$. A* concrete unifier *of terms t_1, t_2 is any valuation $\rho : vars(t_1) \cup vars(t_2) \to \mathcal{T}$ such that $\rho(t_1) =$*

$\rho(t_2)$. *A symbolic unifier* σ *of two terms* t_1, t_2 *is a* most general unifier of t_1, t_2 *with respect to concrete unification whenever, for all concrete unifiers* ρ *of* t_1 *and* t_2*, there is a concrete unifier* $\eta : Z \to \mathcal{T}$ *such that* $\eta \circ \sigma = \rho$.

We often call a symbolic unifier satisfying the above a *most general unifier*, even though the standard notion of most general unifier in algebraic specifications/rewriting is slightly different. We say that terms t_1, t_2 are symbolically (resp. concretely) unifiable if they have a symbolic (resp. concrete) unifier.

Example 3.2.1 *The term* $t_1 \triangleq \texttt{if\_then\_else\_}(B, S_1, S_2)$ *and the term* $t_2 \triangleq \texttt{if\_then\_else\_}(B', S_1, S_2')$ *are symbolically unifiable, by the substitution* $B \mapsto B', S_2 \mapsto S_2'$*, extended to the identity for the other variables occurring in the terms. Since* $\texttt{if\_then\_else\_}(B, S_1, S_2)$ *is a non data function symbols in* Σ *(statements are not data), the two terms are also concretely unifiable, e.g., by any valuation that maps* B *to true and* S_1*,* S_2 *to* $\{\}$ *(empty block).*

In the rest of the section we prove that unification of two given terms, say t_1 and t_2, can be achieved by matching. However, this is possible considering some assumptions over t_1 and t_2. One of these assumptions is that t_1 is *linear*, that is, every variable from $vars(t_1)$ occurs only once in t_1. Another assumption is that every subterm of sort data of t_1 is a variable and all the elements of $vars(t_2)$ have data sorts. The terms we deal with are ML patterns of the form $\pi \wedge \phi$. Every pattern of this form can be transformed such that it is linear and all its data subterms are variables. The transformation consists in replacing the variables and the data subterms with fresh variables and adding equalities between the fresh variables and the terms they replaced to ϕ. Moreover, the transformation does not change the transition system generated by the language (which can be proved by applying Lemma 2.4.2 repeatedly to all subterms of sort data and duplicate variables).

The following lemma shows that for two terms t_1 and t_2, which are concretely unifiable and satisfy the assumptions above, there exists the most general unifier of t_1 and t_2. Moreover, this unifier is obtained via a substitution which represents the matching of t_2 against t_1. In other words, the lemma shows that we can implement symbolic execution (defined in terms of unification) using rewriting.

Lemma 3.2.1 (Unification by Matching) *If t_1 and t_2 are terms such that*

(h_1) $vars(t_1) \cap vars(t_2) = \emptyset$,

(h_2) t_1 *is linear, has a non-data sort, and all its data subterms are variables,*

(h_3) *all the elements of* $vars(t_2)$ *have data sorts, and*

(h_4) t_1, t_2 *are concretely unifiable,*

then there exists a substitution $\sigma : vars(t_1) \to T_\Sigma(vars(t_2))$ *such that* $\sigma(t_1) = t_2$ *and* $\sigma^{t_1}_{t_2} \triangleq \sigma \uplus 1_{vars(t_2)}$ *is a most-general unifier of* t_1, t_2.

Proof By assumptions (h_2), (h_3), and (h_4), the term t_2 has the following properties:

(c_1) t_2 has a non-data sort, and

(c_2) t_2 cannot be a variable.

Indeed, because t_1 and t_2 are concretely unifiable (cf. (h_4)), so there is $\rho : vars(t_1) \cup vars(t_2) \to \mathcal{T}$ of t_1 and t_2, such that $\rho(t_1) = \rho(t_2)$. Since the carrier sets in $\mathcal{T}$ are disjoint then t_1 and t_2 have the same sort, and thus, t_2 has a non-data sort (cf. (h_2)). The term t_2 cannot be a variable because t_2 should have simultaneously a non-data sort (cf. (c_1)), and a data sort, since $t_2 \in vars(t_2)$ (cf. (h_3)). In the rest of the proof we assume (c_1) and (c_2).

We proceed by induction on the structure of t_1. In the base case t_1 is either a variable or a constant:

- t_1 is a variable. In this case we have $vars(t_1) = \{t_1\}$, where t_1 is a variable of non-data sort (cf. (h_2)). We consider $\sigma : vars(t_1) \to T_\Sigma(vars(t_2))$ such that $\sigma(t_1) = t_2$, and $\sigma^{t_1}_{t_2} : vars(t_1) \cup vars(t_2) \to T_\Sigma(vars(t_2))$ where $\sigma^{t_1}_{t_2} \triangleq \sigma \uplus 1_{vars(t_2)}$. Since $\sigma^{t_1}_{t_2}(t_1) = \sigma(t_1) = t_2 = 1_{vars(t_2)}(t_2) = \sigma^{t_1}_{t_2}(t_2)$ then $\sigma^{t_1}_{t_2}$ is a symbolic unifier of t_1, t_2. We have to show that $\sigma^{t_1}_{t_2}$ is also the most general unifier of t_1, t_2. Let us consider any concrete unifier, say, $\rho : vars(t_1) \cup vars(t_2) \to \mathcal{T}$ of t_1 and t_2. Then, $\rho(t_1) = \rho(t_2)$ (cf. Definition 3.2.1) and

$\rho(t_2) = \rho(\sigma_{t_2}^{t_1}(t_1))$ (where $\sigma_{t_2}^{t_1}(t_1) = t_2$ as shown above). Thus, $\rho(t_1) = \rho(t_2) = \rho(\sigma_{t_2}^{t_1}(t_1))$ and for all variables $x \in vars(t_2)$ we have $\rho(x) = \rho(1_{vars(t_2)}(x)) = \rho(\sigma_{t_2}^{t_1}(x))$. Therefore, for all $x \in vars(t_1) \cup vars(t_2) (= \{t_1\} \cup vars(t_2))$, $\rho(\sigma_{t_2}^{t_1}(x)) = \rho(x)$, which proves the fact that $\sigma_{t_2}^{t_1}$ is a most general unifier by taking $\eta = \rho$ in Definition 3.2.1.

- t_1 is a constant symbol. Since t_2 cannot be a variable (cf. (c_2)) then t_2 is either a constant of non-data sort, where $t_1 = t_2$, or t_2 is of the form $g(t_2^1, \ldots, t_2^m)$, where $m > 0$.

  We consider the case when t_2 is a constant. We choose σ the unique function $\emptyset \to T_\Sigma$ extended to terms and we have $\sigma(t_1) = t_1 = t_2$. Moreover, $\sigma_{t_2}^{t_1} \triangleq \sigma \uplus 1_{vars(t_2)} = \sigma$ is a symbolic unifier for t_1 and t_2, and it is also the most general unifier if we choose η the unique function $\emptyset \to \mathcal{T}$ in Definition 3.2.1.

  Now, we consider $t_2 = g(t_2^1, \ldots, t_2^m)$, where $m > 0$. From the fact that ρ is a concrete unifier, we have $\rho(t_1) = \rho(t_2)$. Thus, $\rho(t_1) = \mathcal{T}_f =_{\mathcal{T}} \mathcal{T}_g(t_2^1, \ldots, t_2^m) = \rho(g(t_2^1, \ldots, t_2^m))$, where $=_{\mathcal{T}}$ is the equality in $\mathcal{T}$. Since, t_1 and t_2 have non-data sorts (cf. (h_2) and (c_1)) and $\mathcal{T}$ is a $\mathcal{D}$-grounded model, then according to Definition 3.1.2 we have that $f = g$, $m = 0$. This is a contradiction, since we assumed $m > 0$. Therefore, this case is impossible.

For the inductive step, let $t_1 = f(t_1^1, \ldots, t_1^n)$ with $f \in \Sigma \setminus \Sigma^{\mathfrak{d}}$, $n \geq 1$, and $t_1^1, \ldots, t_1^n \in T_\Sigma(Var)$. There are two subcases regarding t_2:

- t_2 is a constant. This is impossible, since t_1 is not a constant. The proof is similar with the one in the base case, where we considered t_1 as being constant and t_2 not a constant.

- $t_2 = g(t_2^1, \ldots, t_2^m)$ with $g \in \Sigma$, $m \geq 1$, and $t_2^1, \ldots, t_2^m \in T_\Sigma(Var)$. Since ρ is a concrete unifier of t_1, t_2 we have $\rho(t_1) = \rho(f(t_1^1, \ldots, t_1^n)) = \mathcal{T}_f(\rho(t_1^1), \ldots, \rho(t_1^n)) =_{\mathcal{T}} \mathcal{T}_g(\rho(t_2^1), \ldots, \rho(t_2^m)) = \rho(t_2)$, where we emphasize by subscripting the equality symbol with $\mathcal{T}$ that the equality is that of the model $\mathcal{T}$.

  Since $\mathcal{T}$ is a $\mathcal{D}$-grounded model we have $f = g$, $m = n$, and $\rho(t_1^i) = \rho(t_2^i)$ for $i \in \{1, \ldots, n\}$ (cf. Definition 3.1.2). The respective

subterms t_1^i and t_2^i of t_1 and t_2 satisfy the hypotheses of our lemma, except maybe for the fact that t_1^i may have a data sort. There are again two cases:

- if for some $i \in \{1, \ldots, n\}$, t_1^i has a data sort then by (h_2) t_1^i is a variable, with $vars(t_1^i) = \{t_1^i\}$, and we let $\sigma^i : vars(t_1^i) \to T_\Sigma(vars(t_2^i))$ such that $\sigma^i(t_1^i) = t_2^i$. Then, $\sigma_{t_2^i}^{t_1^i} \triangleq \sigma^i \uplus 1_{vars(t_2^i)}$ is a most-general unifier of t_1^i and t_2^i, which is proved like in the base case;
- otherwise, t_1^i and t_2^i satisfy all the the hypotheses of our lemma. We can then use the induction hypothesis and obtain substitutions $\sigma^i : vars(t_1^i) \to T_\Sigma(vars(t_2^i))$ such that $\sigma^i(t_1^i) = t_2^i$ for all $i \in \{1, \ldots, n\}$, and the corresponding most-general-unifiers $\sigma_{t_2^i}^{t_1^i}$ for t_1^i and t_2^i, of the form $\sigma_{t_2^i}^{t_1^i} = \sigma^i \uplus 1_{vars(t_2^i)}$.

Let $\sigma \triangleq \biguplus_{i=1}^n \sigma^i : vars(t_1) \to T_\Sigma(vars(t_2))$, which is a well-defined substitution because t_1 is linear. If $\sigma_{t_2}^{t_1} \triangleq \sigma \uplus 1_{vars(t_2)}$ then $\sigma_{t_2}^{t_1}$ is a symbolic unifier of t_1, t_2:

$$\begin{array}{rcl}
\sigma_{t_2}^{t_1}(t_1) & = & \sigma_{t_2}^{t_1}(f(t_1^1, \ldots, t_1^n)) \\
& = & f(\sigma_{t_2}^{t_1}(t_1^1), \ldots, \sigma_{t_2}^{t_1}(t_1^n)) \\
& = & f(\sigma(t_1^1), \ldots, \sigma(t_1^n)) \\
& = & f(\sigma^1(t_1^1), \ldots, \sigma^n(t_1^n)) \\
& = & f(t_2^1, \ldots, t_2^n) \\
& = & f(1_{vars(t_2)}(t_2^1), \ldots, 1_{vars(t_2)}(t_2^n)) \\
& = & f(\sigma_{t_2}^{t_1}(t_2^1), \ldots, \sigma_{t_2}^{t_1}(t_2^n)) \\
& = & \sigma_{t_2}^{t_1}(f(t_2^1, \ldots, t_2^n)) \\
& = & \sigma_{t_2}^{t_1}(t_2)
\end{array}$$

Next, we prove that $\sigma_{t_2}^{t_1}$ is the most general unifier. We extend $\sigma_{t_2^i}^{t_1^i}$ to $\mu_{t_2}^{t_1^i} : vars(t_1^i) \cup vars(t_2) \to vars(t_2)$ where $\mu_{t_2}^{t_1^i} \triangleq \sigma^i \uplus 1_{vars(t_2)}$ (which is well-defined cf. (h_1), (h_2)). Then:

$$\begin{array}{rcl}\sigma_{t_2}^{t_1} & = & \sigma \uplus 1_{vars(t_2)} \\ & = & (\biguplus_{i=1}^{n} \sigma^i) \uplus 1_{vars(t_2)} \\ & = & \biguplus_{i=1}^{n} (\sigma^i \uplus 1_{vars(t_2)}) \\ & = & \biguplus_{i=1}^{n} \mu_{t_2}^{t_1^i}.\end{array}$$

Since ρ is a concrete unifier of t_1 and t_2, $\rho(t_1^i) = \rho(t_2^i)$ for $i \in \{1, \ldots, n\}$. From the inductive hypothesis we have that each $\sigma_{t_2^i}^{t_1^i}$ is the most-general-unifier of t_1^i and t_2^i for $i = \{1, \ldots, n\}$, so we obtain the existence of valuations $\eta^i : vars(t_2^i) \to \mathcal{T}$ such that $\eta^i \circ \sigma_{t_2^i}^{t_1^i} = \rho|_{vars(t_1^i) \cup vars(t_2^i)}$, for $i = 1, \ldots, n$. We choose $\eta : vars(t_2) \to \mathcal{T}$ such that $\eta(x) = \eta^i(x)$ when $x \in vars(t_2^i)$. Remember that t_2 may not be linear, so for $i \neq j$ we may have $y \in vars(t_2^i) \cap vars(t_2^j)$. Thus, we analyse the cases when $y \in vars(t_2^i)$ and $y \in vars(t_2^j)$:

- $\eta(y) = \eta^i(y) = (\eta^i \circ \sigma_{t_2^i}^{t_1^i})(y) = \rho(y)$, or
- $\eta(y) = \eta^j(y) = (\eta^j \circ \sigma_{t_2^j}^{t_1^j})(y) = \rho(y)$.

We used the fact that $\sigma_{t_2^i}^{t_1^i}$ and $\sigma_{t_2^i}^{t_1^i}$ are defined over $vars(t_1^i)$ and $vars(t_1^j)$, respectively. Either way $\eta(y) = \eta^i(y) = \eta^j(y)$, where from we conclude that η is well-defined.

Let $x \in vars(t_1) \cup vars(t_2)$. Then there is i such that $x \in vars(t_1^i)$ (since t_1 linear). For η and $\sigma_{t_2}^{t_1}$ defined as above we have $(\eta \circ \sigma_{t_2}^{t_1})(x) = \eta(\sigma_{t_2}^{t_1}(x)) = \eta(\mu_{t_2}^{t_1^i}(x)) = \eta(\sigma^i(x)) = \eta^i(\sigma^i(x)) = (\eta^i \circ \sigma^i)(x) = \rho|_{vars(t_1^i) \cup vars(t_2^i)}(x) = \rho(x)$. Thus, η has the property $\eta \circ \sigma_{t_2}^{t_1} = \rho$, which proves that $\sigma_{t_2}^{t_1}$ is a most general unifier of t_1 and t_2 and concludes the proof.

■

Remark 3.2.1 *The most general unifier $\sigma_{t_2}^{t_1}$ whose existence is stated by Lemma 3.2.1 is also unique, because $\sigma_{t_2}^{t_1}$ is defined to be $\sigma \uplus 1_{vars(t_2)}$ and σ, which is a (syntactical) match of t_1 on t_2, is unique when it exists.*

3.3 Symbolic transition relation and properties

In this section we present a symbolic execution approach for languages defined using the language-definition framework presented in Section 3.1. We prove that the transition system generated by symbolic execution forward-simulates the one generated by concrete execution, and that the transition system generated by concrete execution backward-simulates the one generated by symbolic execution (restricted to satisfiable patterns). These properties are the naturally expected ones from a symbolic execution framework. They allow to perform analyses on symbolic programs, and to transfer the results of those analyses to concrete instances of the symbolic programs in question.

In the rest of the section we let $\mathcal{L} \triangleq ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$ be a language definition. Moreover, we assume that $\mathit{vars}(\varphi \Rightarrow \varphi') = \mathit{vars}(\varphi) \cup \mathit{vars}(\varphi')$, and for every rule $\varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ with $\varphi_1 \triangleq \pi_1 \wedge \phi_1$, π_1 satisfies the hypotheses regarding t_1 of Lemma 3.2.1, that is, π_1 is linear, has non-data sort, and all its data subterms are variables. Recall that for patterns $\varphi \triangleq \pi \wedge \phi$ we have $\mathit{vars}(\varphi) = \mathit{vars}(\pi) \cup \mathit{vars}(\phi)$ (Section 2.3). For simplicity, from now on we will use $\mathit{vars}(\pi, \phi)$ to denote $\mathit{vars}(\pi) \cup \mathit{vars}(\phi)$, and for any patterns φ and φ' we let $\mathit{vars}(\varphi, \varphi') \triangleq \mathit{vars}(\varphi_1) \cup \mathit{vars}(\varphi_2)$.

Symbolic execution essentially consists of applying the semantical rules over patterns using the most general unifiers whose existence and unicity are given by Lemma 3.2.1 and Remark 3.2.1.

We first define the relation $\sim$ between patterns:

Definition 3.3.1 *Two patterns φ and φ' are in relation $\sim$, i.e. $\varphi \sim \varphi'$, iff $[\![\varphi]\!] = [\![\varphi']\!]$.*

According to Definition 3.3.1, two patterns φ and φ' are in relation $\sim$ if they are matched by the same set of configurations, i.e. $\{\gamma \mid (\exists \rho)(\gamma, \rho) \models_{\text{ML}} \varphi\}$ and $\{\gamma \mid (\exists \rho)(\gamma, \rho) \models_{\text{ML}} \varphi'\}$ are equal. Therefore, $\sim$ is an equivalence relation. We let $[\varphi]_\sim$ denote the equivalence class of φ. The symbolic transition relation is a relation between such equivalence classes:

Definition 3.3.2 (Symbolic transition relation) *We define the symbolic transition relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ by: $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ iff there is $\widetilde{\varphi} \triangleq \pi \wedge \phi$, such that $\widetilde{\varphi} \sim \varphi$ and all the variables in $\mathit{vars}(\pi)$ have data sorts, and there is a rule $\alpha \triangleq \varphi_1 \Rightarrow \varphi_2 \in \mathcal{S}$ with $\varphi_i \triangleq \pi_i \wedge \phi_i$ for $i \in \{1, 2\}$,*

$vars(\widetilde{\varphi}) \cap vars(\alpha) = \emptyset$, π_1 and π are concretely unifiable, and $[\varphi']_\sim = [\sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)]_\sim$, where $\sigma_\pi^{\pi_1}$ is the most general symbolic unifier of π, π_1 (cf. Lemma 3.2.1), extended as the identity substitution over the variables in $vars(\phi_1, \varphi_2) \setminus vars(\pi, \pi_1)$.

The condition $vars(\varphi) \cap vars(\alpha) = \emptyset$ is not a real restriction because we can always use variable renaming for rules in $\mathcal{S}$.

We call *symbolic rewriting* the manner in which rules are applied in Definition 3.3.2.

Example 3.3.1 *Recall the rule for division from the semantics of CinK shown in Example 3.1.5. The left hand-side of the rule is linear and all its subterms of data sorts are variables. Let us consider a pattern* $\varphi \triangleq \langle\langle X \diagup Y \curvearrowright \cdot\rangle_{\mathsf{k}} \langle Y' \mapsto L\rangle_{\mathsf{env}} \langle L' \mapsto A'\rangle_{\mathsf{store}} \langle S\rangle_{\mathsf{stack}} \langle R\rangle_{\mathsf{return}} \langle I\rangle_{\mathsf{in}} \langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \wedge Y' =_{Int} Y \wedge L' =_{Int} L \wedge A' =_{Int} A +_{Int} 1 \wedge A \neq_{Int} -1$. *Note that φ is linear and all its data subterms are variables. The division rule generates a symbolic transition from $[\varphi]_\sim$ to $[\varphi']_\sim$, where the pattern φ' is* $\langle\langle X \diagup_{Int} Y \curvearrowright \cdot\rangle_{\mathsf{k}} \langle Y' \mapsto L\rangle_{\mathsf{env}} \langle L' \mapsto A'\rangle_{\mathsf{store}} \langle S\rangle_{\mathsf{stack}} \langle R\rangle_{\mathsf{return}} \langle I\rangle_{\mathsf{in}} \langle O\rangle_{\mathsf{out}}\rangle_{\mathsf{cfg}} \wedge Y' =_{Int} Y \wedge L' =_{Int} L \wedge A' =_{Int} A +_{Int} 1 \wedge A \neq_{Int} -1 \wedge Y \neq_{Int} 0$.

Assumption 3.3.1 *From now on we assume that for all elementary patterns $\varphi \triangleq \pi \wedge \phi$, $\varphi' \triangleq \pi' \wedge \phi'$ such that $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi']_\sim$, π and π' may only have variables of data sorts. This can be obtained by starting with an initial pattern satisfying these assumptions and by ensuring that they are preserved by the rules $\mathcal{S}$. Thus, our symbolic execution framework allows symbolic data but no symbolic code.*

We now show that the symbolic execution thus defined is related with concrete execution via the coverage and precision properties stated in the introduction of this chapter.

3.3.1 Coverage

The coverage property states that the symbolic transition system forward-simulates the concrete transition system.

Lemma 3.3.1 (Coverage) *If $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\varphi]\!]$ (with all variables in $vars(\varphi)$ of data sorts) then there exists φ' such that $\gamma' \in [\![\varphi']\!]$ and $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathfrak{s}} [\varphi']_\sim$.*

Proof Let $\varphi \triangleq \pi \wedge \phi$. From $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ we obtain the rule $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ and the valuation $\rho : Var \to \mathcal{T}$ such that $\gamma = \rho(\pi_1)$, $\rho \models_{\mathrm{ML}} \phi_1$, $\gamma' = \rho(\pi_2)$, and $\rho \models_{\mathrm{ML}} \phi_2$. From $\gamma \in [\![\varphi]\!]$ we obtain the valuation $\mu : Var \to \mathcal{T}$ such that $\gamma = \mu(\pi)$ and $\mu \models_{\mathrm{ML}} \phi$. We assume, without loss of generality, that $vars(\varphi) \cap vars(\alpha) = \emptyset$ (which can always be obtained by variable renaming). The basic patterns π_1 and π are concretely unifiable if we choose $\rho|_{vars(\pi_1)} \uplus \mu|_{vars(\pi)}$ as their concrete unifier. Then, using Lemma 3.2.1 we obtain their unique most-general symbolic unifier $\sigma_\pi^{\pi_1}$, whose codomain is $T_\Sigma(vars(\pi_1) \cup vars(\pi))$. Let $\eta : vars(\pi_1) \cup vars(\pi) \to \mathcal{T}$ be the valuation such that $\eta \circ \sigma_\pi^{\pi_1} = \rho|_{vars(\pi_1)} \uplus \mu|_{vars(\pi)}$ (cf. Definition 3.2.1). We extend $\sigma_\pi^{\pi_1}$ to $vars(\varphi) \cup vars(\alpha)$ by letting it be the identity on $(vars(\varphi) \cup vars(\alpha)) \setminus vars(\pi_1, \pi)$, and extend η to $vars(\varphi) \cup vars(\alpha)$ such that $\eta|_{vars(\phi_1,\pi_2,\phi_2)\setminus vars(\pi_1)} = \rho|_{vars(\phi_1,\pi_2,\phi_2)\setminus vars(\pi_1)}$ and $\eta|_{vars(\phi)\setminus vars(\pi)} = \mu|_{vars(\phi)\setminus vars(\pi)}$. With these extensions we have $(\eta \circ \sigma_\pi^{\pi_1})(x) = (\rho|_{vars(\alpha)} \uplus \mu|_{vars(\varphi)})(x)$ for all $x \in vars(\varphi) \cup vars(\alpha)$.

Let $\varphi' \triangleq \sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$. Then, according to Definition 3.3.2 we have the transition $[\varphi]_\sim \Rightarrow_{\mathcal{S}}^{\mathsf{s}} [\varphi']_\sim$. There remains to prove $\gamma' \in [\![\varphi']\!]$.

- On the one hand, $\eta(\sigma_\pi^{\pi_1}(\pi_2)) = (\eta \circ \sigma_\pi^{\pi_1})(\pi_2) = (\rho \uplus \mu)(\pi_2) = \rho(\pi_2) = \gamma'$; thus, there is a valuation η such that $(\gamma', \eta) \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\pi_2)$.

- On the other hand,

$$\begin{aligned} \eta &\models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2) && \text{iff} \\ \eta \circ \sigma_\pi^{\pi_1} &\models_{\mathrm{ML}} (\phi \wedge \phi_1 \wedge \phi_2) && \text{iff} \\ \rho \uplus \mu &\models_{\mathrm{ML}} (\phi \wedge \phi_1 \wedge \phi_2) && \text{iff} \\ \mu \models_{\mathrm{ML}} \phi \text{ and } \rho &\models_{\mathrm{ML}} \phi_1 \text{ and } \rho \models_{\mathrm{ML}} \phi_2. \end{aligned}$$

  Since the last relations hold by the hypotheses, it follows $\eta \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$. Note that we used the property: if $\rho : Var \to \mathcal{T}$ is a valuation and $\sigma : Var \to T_\Sigma(Var)$ a substitution, then $\rho \models_{\mathrm{ML}} \sigma(\varphi)$ iff $\rho \circ \sigma \models_{\mathrm{ML}} \varphi$.

The two above items imply $(\gamma', \eta) \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$, i.e., $(\gamma', \eta) \models_{\mathrm{ML}} \varphi'$, which concludes the proof.
■

Remark 3.3.1 *Note that in the proof of the Lemma 3.3.1, the symbolic transition* $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ *is given by a rule which corresponds (cf. Definition 3.3.2) to the rule which gives the concrete transition* $\gamma \Rightarrow_{\mathcal{S}} \gamma'$.

The next corollary follows immediately from Lemma 3.3.1. It can be used to draw conclusions about the absence of concrete program executions on a given path from the absence of feasible symbolic executions on the same path.

Corollary 3.3.1 *For every concrete execution* $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$, *and pattern* φ_0 *such that* $\gamma_0 \in [\![\varphi_0]\!]$ *(with all the variables in* $vars(\varphi_0)$ *of data sorts), there exists a symbolic execution* $[\varphi_0]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_1]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_n]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ *such that* $\gamma_i \in [\![\varphi_i]\!]$ *for* $i = 0, 1, \ldots$.

Proof From the hypothesis we have $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ and φ_0 such that $\gamma_0 \in [\![\varphi_0]\!]$ (with all the variables in $vars(\varphi_0)$ of data sorts). Then, for each $\gamma_i \Rightarrow_{\mathcal{S}} \gamma_{i+1}$, where $i \in \{0, 1, \ldots\}$ we have $[\varphi_i]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_{i+1}]_\sim$ and $\gamma_{i+1} \in [\![\varphi_{i+1}]\!]$ (cf. Lemma 3.3.1).
■

Remark 3.3.2 *Corollary 3.3.1 states that for every concrete execution there exists a symbolic one. Moreover, symbolic execution takes the same path in the program as concrete execution, since there is a one to one correspondence between the rules that generate symbolic and concrete transitions (cf. Remark 3.3.1).*

3.3.2 Precision

The precision property states that the symbolic transition system is backwards simulated by the concrete one. Forward simulation could not hold in this case, because the patterns resulting from a symbolic transition may be unsatisfiable. A pattern φ is satisfiable if there is a configuration γ such that $\gamma \in [\![\varphi]\!]$; otherwise, φ is unsatisfiable.

Lemma 3.3.2 (Precision) *If* $\gamma' \in [\![\varphi']\!]$ *and* $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ *then there exists* $\gamma \in \mathcal{T}_{Cfg}$ *such that* $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ *and* $\gamma \in [\![\varphi]\!]$.

Proof Let $\widetilde{\varphi} \triangleq \pi \wedge \phi$ such that $\widetilde{\varphi} \sim \varphi$ (as shown in Definition 3.3.2). Since $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ then (by Definition 3.3.2) there is a rule $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow$

$\pi_2 \wedge \phi_2 \in \mathcal{S}$ such that π_1 and π are concretely unifiable, and $[\varphi']_\sim = [\sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)]_\sim$, where $\sigma_\pi^{\pi_1}$ is the most general unifier (cf. Lemma 3.2.1). Thus, $\varphi' \sim \sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$, that is $[\![\varphi']\!] = [\![\sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)]\!]$. From $\gamma' \in [\![\varphi']\!]$ and the previous equality we obtain $\eta : \mathit{Var} \to \mathcal{T}$ such that $\gamma' = \eta(\sigma_\pi^{\pi_1}(\pi_2))$ and $\eta \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$.

We extend $\sigma_\pi^{\pi_1}$ to $\mathit{vars}(\varphi) \cup \mathit{vars}(\alpha)$ by letting it be the identity on $(\mathit{vars}(\widetilde{\varphi}) \cup \mathit{vars}(\alpha)) \setminus \mathit{vars}(\pi_1, \pi)$. Let $\rho : \mathit{Var} \to \mathcal{T}$ be defined by $\rho(x) = (\eta \circ \sigma_\pi^{\pi_1})(x)$ for all $x \in \mathit{vars}(\widetilde{\varphi}) \cup \mathit{vars}(\pi_1)$, and $\rho(x) = \eta(x)$ for all $x \in \mathit{Var} \setminus (\mathit{vars}(\widetilde{\varphi}) \cup \mathit{vars}(\pi_1))$. Let $\gamma \triangleq \rho(\pi_1)$. We have $\gamma' = \eta(\sigma_\pi^{\pi_1}(\pi_2)) = \eta(\pi_2) = \rho(\pi_2)$. From $\eta \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$ we get $\eta \circ \sigma_\pi^{\pi_1} \models_{\mathrm{ML}} \phi \wedge \phi_1 \wedge \phi_2$. In particular, $\eta \circ \sigma_\pi^{\pi_1} \models_{\mathrm{ML}} \phi_1$ and $\eta \circ \sigma_\pi^{\pi_1} \models_{\mathrm{ML}} \phi_2$, i.e., $\rho \models_{\mathrm{ML}} \phi_1$ and $\rho \models_{\mathrm{ML}} \phi_2$. Since $\gamma \triangleq \rho(\pi_1)$, $\gamma' = \rho(\pi_2)$ and $\alpha \in \mathcal{S}$ then there is a transition $\gamma \Rightarrow_{\mathcal{S}} \gamma'$.

There remains to prove $\gamma \in [\![\varphi]\!] (= [\![\widetilde{\varphi}]\!])$.

- From $\gamma = \rho(\pi_1)$ using the definition of ρ we get $\gamma = \rho(\pi_1) = (\eta \circ \sigma_\pi^{\pi_1})(\pi_1) = \eta(\sigma_\pi^{\pi_1}(\pi_1)) = \eta(\sigma_\pi^{\pi_1}(\pi)) = (\eta \circ \sigma_\pi^{\pi_1})(\pi) = \rho(\pi)$
- From $\eta \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$ and $\eta \models_{\mathrm{ML}} \sigma_\pi^{\pi_1}(\pi_1)$ iff $\eta \circ \sigma_\pi^{\pi_1} \models_{\mathrm{ML}} \phi$ we get $\rho \models_{\mathrm{ML}} \phi$.

Since $\varphi \triangleq \pi \wedge \phi$, the items above imply $(\gamma, \rho) \models \varphi$, i.e., $\gamma \in [\![\varphi]\!]$, which completes the proof.
■

Remark 3.3.3 *Note that in the proof of Lemma 3.3.2 the rule that generates the concrete transition $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ corresponds (cf. Definition 3.3.2) to the rule that generates the symbolic transition $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$.*

We call a symbolic execution *feasible* if all its patterns are satisfiable. The next corollary, which follows from Lemma 3.3.2, can be used to draw conclusions on the existence of concrete program executions on a given path from the existence of feasible symbolic executions on the same path.

Corollary 3.3.2 *For every feasible symbolic execution $[\varphi_0]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_1]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_n]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ there is a concrete execution $\gamma_0 \Rightarrow_{\mathcal{S}} \gamma_1 \Rightarrow_{\mathcal{S}} \cdots \Rightarrow_{\mathcal{S}} \gamma_n \Rightarrow_{\mathcal{S}} \cdots$ such that $\gamma_i \in [\![\varphi_i]\!]$ for $i = 0, 1, \ldots$.*

Proof Since $[\varphi_0]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_1]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi_n]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} \cdots$ is feasible then for each φ_i, $i \in \{0, 1, \ldots\}$, there is $\gamma_i \in [\![\varphi_i]\!]$. Then, applying Lemma 3.3.2 we choose γ_i such that there is a transition $\gamma_i \Rightarrow_{\mathcal{S}} \gamma_{i+1}$ for $i \in \{0, 1, \ldots\}$ and $\gamma_i \in [\![\varphi_i]\!]$.
■

Remark 3.3.4 *Corollary 3.3.2 states that for every feasible execution there is a corresponding concrete execution. Moreover, the execution follows the same path in the program, since there is a one to one correspondence between the rules that generate transitions (cf. Remark 3.3.3).*

The corollaries in this section say that symbolic execution can be used as a sound program-analysis technique. However, symbolic execution is, in general, not powerful enough for performing program verification, because one can (obviously) only generate bounded-length symbolic executions, whereas program verification, especially in the presence of loops and recursive function calls, would require in general executions of an unbounded length.

Remark 3.3.5 *Let $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ be the only rule in $\mathcal{S}$ and patterns $\varphi \triangleq \pi \wedge \phi$, φ', $\widehat{\varphi} \triangleq \widehat{\pi} \wedge \widehat{\phi}$, $\widehat{\varphi}'$ such that $\varphi \sim \widehat{\varphi}$, $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$, and $[\widehat{\varphi}]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\widehat{\varphi}']_\sim$, where π_1 and π are concretely unifiable, $[\varphi']_\sim \triangleq [\sigma^{\pi_1}_{\pi}(\pi_2) \wedge \sigma^{\pi_1}_{\pi}(\phi \wedge \phi_1 \wedge \phi_2)]_\sim$, π_1 and $\widehat{\pi}$ are concretely unifiable, and $[\widehat{\varphi}']_\sim \triangleq [\sigma^{\pi_1}_{\widehat{\pi}}(\pi_2) \wedge \sigma^{\pi_1}_{\widehat{\pi}}(\widehat{\phi} \wedge \phi_1 \wedge \phi_2)]_\sim$.*

Let $\gamma' \in [\![\varphi']\!]$. From $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ and Lemma 3.3.2 there is γ such that $\gamma \in [\![\varphi]\!]$ and $\gamma \Rightarrow_{\mathcal{S}} \gamma'$. Since $\varphi \sim \widehat{\varphi}$ we also obtain $\gamma \in [\![\widehat{\varphi}]\!]$. By Lemma 3.3.1 with $\gamma \Rightarrow_{\mathcal{S}} \gamma'$ and $\gamma \in [\![\widehat{\varphi}]\!]$ as hypotheses, there exists $\widehat{\varphi}''$ such that $[\widehat{\varphi}]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\widehat{\varphi}'']_\sim$ and $\gamma' \in [\widehat{\varphi}'']_\sim$. In the proof of Lemma 3.3.1, $\widehat{\varphi}''$ is chosen as $\sigma^{\pi_1}_{\widehat{\pi}}(\pi_2) \wedge \sigma^{\pi_1}_{\widehat{\pi}}(\widehat{\phi} \wedge \phi_1 \wedge \phi_2)$, and thus, $[\![\widehat{\varphi}'']\!] = [\![\sigma^{\pi_1}_{\widehat{\pi}}(\pi_2) \wedge \sigma^{\pi_1}_{\widehat{\pi}}(\widehat{\phi} \wedge \phi_1 \wedge \phi_2)]\!] = [\![\widehat{\varphi}']\!]$. Therefore, $\gamma' \in [\![\widehat{\varphi}']\!]$.

Conversely, we can show that if $\gamma' \in [\![\widehat{\varphi}']\!]$ then $\gamma' \in [\![\varphi']\!]$ using a similar approach. Therefore, $[\![\varphi']\!] = [\![\widehat{\varphi}']\!]$, that is $[\varphi']_\sim = [\widehat{\varphi}']_\sim$. In conclusion, for two different patterns φ and $\widehat{\varphi}$ such that $\varphi \sim \widehat{\varphi}$, a rule in $\mathcal{S}$ induces at most one symbolic transition, i.e. $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim (= [\widehat{\varphi}']_\sim)$.

Remark 3.3.6 *The symbolic transition relation is finitely branching since the set $\mathcal{S}$ of rules is finite and cf. Remark 3.3.5 each rule determines at most one symbolic transition.*

The Coverage and Precision properties depend on the conditions exhibited in the hypothesis of Lemma 3.2.1. Essentially, these conditions apply to both the rules in the semantics, and the programs that we run symbolically. In practice, we can perform some mechanical transformations over the rules in the semantics such that their are linear and all data subterms are variables. On the other hand, the condition over programs (i.e. all variables are of data sorts) remains a real restriction, and thus, our symbolic execution framework is not able to run programs where pieces of code are data. Moreover, if the conditions of the Lemma are not met then the Coverage and Precision results are not guaranteed to hold, since the symbolic execution relation depends on the existence of the most general unifier (which also depends on these conditions)

3.4 Symbolic Execution via Language Transformation

In this section we show how to achieve symbolic execution in language-definition frameworks, such as $\mathbb{K}$, where rules are applied by standard rewriting. This forms the basis of our prototype implementation of symbolic execution in the $\mathbb{K}$ framework, which is presented in the Section 5.1.

The symbolic semantics of programming languages (given in Definition 3.3.2) requires rules to be applied in a symbolic manner. A fair question that arises is then: how to implement this symbolic rewriting in a setting where only standard rewriting is available, such as our generic language definition framework, where languages are triples $\mathcal{L} = ((\Sigma, \Pi, \mathit{Cfg}), \mathcal{T}, \mathcal{S})$? The answer is to *transform* a language definition $\mathcal{L}$ into another language definition $\mathcal{L}^{\mathfrak{s}}$, such that standard rewriting in $\mathcal{L}^{\mathfrak{s}}$ corresponds to symbolic rewriting in $\mathcal{L}$.

In the following we define the components of $\mathcal{L}^{\mathfrak{s}}$:

1. $(\Sigma^{\mathfrak{s}}, \Pi^{\mathfrak{s}}, \mathit{Cfg}^{\mathfrak{s}})$. The signature $\Sigma^{\mathfrak{s}}$ is Σ extended with a new sort Cond, with constructors for formulas, and a new sort $\mathit{Cfg}^{\mathfrak{s}}$ (for symbolic configurations) with the constructor $\_\wedge^{\mathfrak{s}}\_ : \mathit{Cfg} \times \mathit{Cond} \to \mathit{Cfg}^{\mathfrak{s}}$. This leaves $(\Sigma^{\mathfrak{s}})^{\mathfrak{d}}$ unchanged, i.e. $(\Sigma^{\mathfrak{s}})^{\mathfrak{d}} = \Sigma^{\mathfrak{d}}$. The predicates in Π are

transformed into terms of sort *Cond*, hence, Π is not included in $\Pi^{\mathfrak{s}}$. Here, we have the freedom to choose $\Pi^{\mathfrak{s}}$ such that $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ corresponds to $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ (cf. Proposition 3.4.1), or to the subset of $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ corresponding to the feasible executions, or to a subset of $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ that approximates the feasible executions. We first choose $\Pi^{\mathfrak{s}} = \emptyset$. For an elementary pattern $\varphi \triangleq \pi \wedge \phi$, let $\varphi^{\mathfrak{s}}$ be the symbolic configuration $\varphi^{\mathfrak{s}} \triangleq \pi \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \phi$, and for each symbolic configuration $\varphi^{\mathfrak{s}} \triangleq \pi \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \phi$, let $\varphi \triangleq \pi \wedge \phi$ be the corresponding elementary pattern.

2. The data domain $\mathcal{D}^{\mathfrak{s}}$ is the set of terms $\bigcup_d T_{\Sigma,d}(\mathit{Var}^{\mathfrak{d}})$, where d is a data sort and $\mathit{Var}^{\mathfrak{d}}$ denotes the subset of variables of data sort. The model $\mathcal{T}^{\mathfrak{s}}$ is then the set of ground terms over the signature $(\Sigma^{\mathfrak{s}} \setminus (\Sigma^{\mathfrak{s}})^{\mathfrak{d}}) \cup \mathcal{D}^{\mathfrak{s}}$.

3. For each rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ in $\mathcal{S}$, the set of rules $\mathcal{S}^{\mathfrak{s}}$ includes a rule of the form $\pi_1 \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \psi \Rightarrow \pi_2 \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} (\psi \wedge \phi_1 \wedge \phi_2)$, where ψ is fresh a variable of sort *Cond*.

The first implication in Proposition 3.4.1 says that for each transition in $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ there is a transition in $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ that is "more general" than the one in $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$, while the second implication says that for each transition in $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ there exists essentially the same transition in $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ (up to equivalent pattern conditions).

Proposition 3.4.1 (Relating $\Rightarrow^{\mathfrak{s}}_{\mathcal{S}}$ and $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$)

1. *If* $\widehat{\varphi}^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \widehat{\varphi'}^{\mathfrak{s}}$ *then there exist a substitution* σ *and patterns* φ *and* φ' *such that* $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ *and* $\widehat{\varphi} = \sigma(\varphi), \widehat{\varphi}' = \sigma(\varphi')$.

2. *If* $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ *then* $\widehat{\varphi}^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \widehat{\varphi'}^{\mathfrak{s}}$ *for some* $\widehat{\varphi} \in [\varphi]_\sim$, $\widehat{\varphi}' \in [\varphi']_\sim$.

Proof We recall that, by Assumption 3.3.1, all the free variables occuring in the basic patterns of φ, φ' (and thus also in $\varphi^{\mathfrak{s}}, \varphi'^{\mathfrak{s}}$ and in all patterns $\sim$-equivalent to φ, φ') are of data sorts. This is used in the proof to apply Lemma 3.2.1.

(1) Assume $\widehat{\varphi}^{\mathfrak{s}} \Rightarrow_{\mathcal{S}^{\mathfrak{s}}} \widehat{\varphi'}^{\mathfrak{s}}$, where $\widehat{\varphi} \triangleq \pi \wedge \phi$. Then, there is a rule $\pi_1 \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \psi \Rightarrow \pi_2 \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} (\psi \wedge \phi_1 \wedge \phi_2) \in \mathcal{S}^{\mathfrak{s}}$, generated from $\alpha \triangleq \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$, and a valuation $\rho^{\mathfrak{s}} : \mathit{Var} \cup \{\psi\} \to \mathcal{T}^{\mathfrak{s}}$ such that $\widehat{\varphi}^{\mathfrak{s}} = \rho^{\mathfrak{s}}(\pi_1 \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \psi)$ and $\widehat{\varphi'}^{\mathfrak{s}} = \rho^{\mathfrak{s}}(\pi_2) \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \rho^{\mathfrak{s}}(\psi \wedge \phi_1 \wedge \phi_2)$. From $\rho^{\mathfrak{s}}(\pi_1 \mathbin{\mathbb{\wedge}^{\mathfrak{s}}} \psi) = \widehat{\varphi}^{\mathfrak{s}}$

we obtain $\pi = \rho^\mathfrak{s}(\pi_1)$ and $\phi = \rho^\mathfrak{s}(\psi)$. We assume, without restriction of generality (possibly, after renaming some variables), that $vars(\widehat{\varphi}^\mathfrak{s}) \cap vars(\pi_1 \wedge \phi_1) = \emptyset$. It follows that the restriction of $\rho^\mathfrak{s}$ to $vars(\pi_1, \pi)$ is the same with the most-general-unifier $\sigma_\pi^{\pi_1}$ given by Lemma 3.2.1.

Let $\varphi \triangleq \widehat{\varphi}$ and $\varphi' \triangleq \sigma_\pi^{\pi_1}(\pi_2 \wedge \phi \wedge \phi_1 \wedge \phi_2)$. We obviously have $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$ by Definition 3.3.2. We recall that $\sigma_\pi^{\pi_1}$ is extended as identity over $vars(\alpha, \widehat{\varphi}^\mathfrak{s}) \setminus vars(\pi, \pi_1)$ in Definition 3.3.2. Therefore, we consider the substitution σ defined by $\sigma(x) = \rho^\mathfrak{s}(x)$ for $x \in vars(\alpha, \widehat{\varphi}^\mathfrak{s}) \setminus vars(\pi, \pi_1)$ and as the identity in the rest. We obtain $\sigma(\varphi') = \widehat{\varphi}'$ and $\sigma(\varphi) = \sigma(\widehat{\varphi}) = \widehat{\varphi}$ by the definition of σ, which concludes the first implication of the proposition.

(2) Assume $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$. Then, there is $\widehat{\varphi} \triangleq \pi \wedge \phi \in [\varphi]_\sim$ and a rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ such that $[\varphi']_\sim = [\sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)]_\sim$, where $\sigma_\pi^{\pi_1}$ is the most general unifier of π and π_1, cf. Lemma 3.2.1. We choose $\widehat{\varphi}' \triangleq \sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)$. Lemma 3.2.1 also says that $\sigma_\pi^{\pi_1}$ can be decomposed into a substitution σ over $vars(\pi_1)$ such that $\sigma(\pi_1) = \pi$, and the identity substitution over $vars(\pi)$. Since $vars(\pi) \cap vars(\pi_1) = \emptyset$ we can obtain, possibly by renaming variables, that $vars(\pi \wedge \phi) \cap vars(\pi_1) = \emptyset$, and thus σ has no effect on ϕ. We obtain $\widehat{\varphi}' = \sigma(\pi_2) \wedge \phi \wedge \sigma(\phi_1 \wedge \phi_2)$. Let then $\rho^\mathfrak{s} : Var \cup \{\psi\} \to \mathcal{T}^\mathfrak{s}$ be any valuation such that $\rho^\mathfrak{s}(x) = \sigma(x)$ for all $x \in vars(\pi_1) \cup vars(\pi_2)$ and $\rho^\mathfrak{s}(\psi) = \phi$. We obtain $\rho^\mathfrak{s}(\pi_1 \mathbin{\mathbf{\Lambda}^\mathfrak{s}} \psi) = \widehat{\varphi}^\mathfrak{s}$ and $\rho^\mathfrak{s}(\pi_2 \mathbin{\mathbf{\Lambda}^\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2)) = \widehat{\varphi}'^\mathfrak{s}$, which means that the transition $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi}'^\mathfrak{s}$ is generated by the rule $\pi_1 \mathbin{\mathbf{\Lambda}^\mathfrak{s}} \psi \Rightarrow \pi_2 \mathbin{\mathbf{\Lambda}^\mathfrak{s}} (\psi \wedge \phi_1 \wedge \phi_2) \in \mathcal{S}^\mathfrak{s}$ and the valuation $\rho^\mathfrak{s}$.

■

A coverage result (cf. Section 3.3.1) relating $\Rightarrow_\mathcal{S}$ and $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ follows from the original coverage result and the second implication of Proposition 3.4.1. Let us assume $\gamma \Rightarrow_\mathcal{S} \gamma'$ and $\gamma \in [\![\widehat{\varphi}^\mathfrak{s}]\!]$, where $[\![\widehat{\varphi}^\mathfrak{s}]\!] \triangleq [\![\widehat{\varphi}]\!]$. Since $\varphi \sim \widehat{\varphi}$ (i.e. $[\![\varphi]\!] = [\![\widehat{\varphi}]\!]$) then $\gamma \in [\![\varphi]\!]$, which, together with $\gamma \Rightarrow_\mathcal{S} \gamma'$, implies, thanks to the original coverage result, that there is φ' such that $\gamma' \in [\![\varphi']\!]$ and $[\varphi]_\sim \Rightarrow^\mathfrak{s}_\mathcal{S} [\varphi']_\sim$. From the second implication of Proposition 3.4.1 we have $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi}'^\mathfrak{s}$ and $\widehat{\varphi}' \in [\varphi']_\sim$ (i.e. $[\![\varphi']\!] = [\![\widehat{\varphi}']\!]$), and thus we obtain $\gamma' \in [\![\widehat{\varphi}'^\mathfrak{s}]\!]$. This gives the coverage of $\Rightarrow_\mathcal{S}$ by $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$.

A precision result relating $\Rightarrow_\mathcal{S}$ by $\Rightarrow_{\mathcal{S}^\mathfrak{s}}$ also follows from the original precision result and the first implication of Proposition 3.4.1, by using the fact (taken from its proof) that $\varphi = \widehat{\varphi}$. Indeed, assume $\widehat{\varphi}^\mathfrak{s} \Rightarrow_{\mathcal{S}^\mathfrak{s}} \widehat{\varphi}'^\mathfrak{s}$

and $\gamma' \in [\![\widehat{\varphi}'^{\mathfrak{s}}]\!]$. Since $\widehat{\varphi}' = \sigma(\varphi')$ it follows that $\gamma' \in [\![\varphi']\!]$ (i.e., φ being more general than $\widehat{\varphi}'$, it contains all the instances of $\widehat{\varphi}'$, including the configuration γ'). Using $[\varphi]_\sim \Rightarrow^{\mathfrak{s}}_{\mathcal{S}} [\varphi']_\sim$ and the original precision result, we obtain a configuration $\gamma \in [\![\varphi]\!]$ such that $\gamma \Rightarrow_{\mathcal{S}} \gamma'$, and since the proof of the first implication gave $\varphi = \widehat{\varphi}$, we have $\gamma \in [\![\widehat{\varphi}^{\mathfrak{s}}]\!]$, proving the precision of $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ with respect to $\Rightarrow_{\mathcal{S}}$.

Remark 3.4.1 *The coverage and precision results relating $\Rightarrow_{\mathcal{S}}$ and $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ are important because they say that $\Rightarrow_{\mathcal{S}^{\mathfrak{s}}}$ has the natural properties expected from symbolic execution.*

The feasible symbolic executions can be obtained as executions of another slightly different definition of $\mathcal{L}^{\mathfrak{s}}$, at least theoretically, by considering $\Pi^{\mathfrak{s}}_{Cond} = \{sat\}$ and $\Pi^{\mathfrak{s}}_{s} = \emptyset$ for all sorts s other than $Cond$, with the interpretation $\mathcal{T}^{\mathfrak{s}}_{sat} = \{\phi \mid \phi \text{ is satisfiable in } \mathcal{T}\}$, and by taking in $\mathcal{S}^{\mathfrak{s}}$ the following conditional rules, for each $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$:

$$(\pi_1 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} \psi) \mathbin{\mathbf{\wedge}} sat(\psi \wedge \phi_1 \wedge \phi_2) \mathbin{\mathbf{\Rightarrow}} \pi_2 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} (\psi \wedge \phi_1 \wedge \phi_2)$$

Recall that $\pi_1 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} \psi$, $\pi_2 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} (\psi \wedge \phi_1 \wedge \phi_2)$ are terms in $\mathcal{L}^{\mathfrak{s}}$, thus, the expression $(\pi_1 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} \psi) \wedge sat(\psi \wedge \phi_1 \wedge \phi_2)$ is a pattern and the above rule is well formed.

However, this definition of $\mathcal{L}^{\mathfrak{s}}$ cannot be implemented in practice because the satisfiability problem for FOL is undecidable. To deal with this issue we implemented a slightly modified version of $\mathcal{L}^{\mathfrak{s}}$ which approximates feasible symbolic executions. This can be done by using a predicate symbol $nsat$ instead of sat, such that its interpretation is sound, i.e., $\mathcal{T}^{\mathfrak{s}}_{nsat} \subsetneq \{\phi \mid \phi \text{ not satisfiable in } \mathcal{T}\}$, and is computable. Then, the rules in $\mathcal{S}^{\mathfrak{s}}$ have the form $(\pi_1 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} \psi) \wedge \neg\, nsat(\psi \wedge \phi_1 \wedge \phi_2) \mathbin{\mathbf{\Rightarrow}} \pi_2 \mathbin{\mathbf{\wedge}^{\mathfrak{s}}} (\psi \wedge \phi_1 \wedge \phi_2)$. The predicate $nsat$ is an approximation of the theoretical $unsat$ predicate. This version of $\mathcal{L}^{\mathfrak{s}}$ is what we actually implemented in our prototype tool in $\mathbb{K}$ Framework, which is presented in Section 5.1.

Chapter 4

Applications

Symbolic execution has been used as a powerful program analysis technique in many applications (Section 1.3). In this dissertation, we focused mainly on using it for program verification. This chapter describes two approaches based on Hoare Logic and Reachability Logic, respectively. In Section 4.1 we present our approach from [4], where we show that, for a given language, we can use our symbolic execution framework to verify Hoare triples. We present a Hoare Logic proof system for IMP, and its equivalent proof system given using reachability rules, obtained by a transformation proposed in [100]. We prove that symbolic execution yields a reachability rule which can be derived with this proof system. By its nature, verification based on Hoare Logic is a bit limited. First, it is not easy to express with Hoare triples properties for programs which involve recursive function calls, blocks with local variables, virtual method calls, etc. Second, Hoare Logic is language dependent, i.e. for every new language construct we may need additional proof rules. On the other hand, our symbolic execution is language independent, and we take advantage of this in Section 4.2, where we propose a language independent proof system, which has symbolic execution as a distinguished proof rule, and a default strategy for applying the proof rules, such that we can prove RL formulas. Formally, we prove that our deductive system together with the default strategy is sound and weakly-complete. An implementation of this proof system and examples are discussed in Section 5.3.

4.1 Hoare-Logic Verification

Hoare Logic [54] (also known as Floyd-Hoare Logic) is probably the most popular formal system for reasoning about the correctness of computer programs. Proposed in 1969 by C. A. R. Hoare, the logic was inspired from the work of R. Floyd on flowcharts [42], and it consists in a set of deduction rules which prove the validity of Hoare triples. Triples are formulas of the form $(\!|\psi|\!)St(\!|\psi'|\!)$ representing program properties. St is a piece of code (statements), while ψ and ψ' are program assertions, known as *precondition* and *postcondition*, respectively. Intuitively, such formulas have the following semantics: a triple $(\!|\psi|\!)St(\!|\psi'|\!)$ holds, w.r.t. the partial correctness, if and only if any terminating execution of St, starting in a state satisfying the precondition ψ, ends in a state satisfying the postcondition ψ'. If further, the precondition ensures the termination of St, then $(\!|\psi|\!)St(\!|\psi'|\!)$ holds w.r.t. total correctness.

In [4] we present a methodology which, applied to the $\mathbb{K}$ definition of a programming language, turns the $\mathbb{K}$ runner into a Hoare-like verifier. The work was inspired from [49], where the authors present methods for proving Hoare formulas based both on the forward computation of postconditions and on the backwards computation of preconditions. According to [49], a Hoare triple can be verified by symbolically executing separate paths and combining the results. Intuitively, we follow the same approach: we first perform symbolic execution starting in a state where the precondition is the initial path condition, and after the execution we verify if the collected path condition implies the postcondition. If the implication holds for all program paths, then the program is proved correct.

To our best knowledge, there are only a few notable Hoare Logic verification approaches based on the formal semantics of a language (e.g. Caduceus [18], Java-ITP [104], KeY-Hoare [43], Spec# [15]). Others (e.g. ITS [50], JAlgo [58]) are not even based on formal definitions of programming languages.

In this section we show that symbolic execution can be used to verify Hoare triples. The approach is based on a mechanical translation of Hoare triples into reachability rules. The obtained reachability rules can be proved by executing symbolically their left hand side and by checking if the resulted patterns imply the right hand side of the rule.

We present the proof system of Hoare Logic [HL-IMP] by means of a simple imperative language, called IMP (Section 4.1.1). This proof system has been shown sound in [100], using a technique which mechanically translates Hoare triples into reachability rules and Hoare Logic proof derivations into Reachability Logic proof derivations. We show this translation in Section 4.1.2, and then we use it to construct the equivalent Hoare Logic proof system [HL'-IMP] given using reachability rules. Then, we extend IMP with syntax and semantics for Hoare-like annotations (preconditions, postconditions, and invariants) and we show that symbolic execution in the annotated language yields a proof derivation in [HL'-IMP] (Section 4.1.3).

$$
\begin{array}{rl}
\mathit{Id} & ::= \text{domain of identifiers} \\
\mathit{Int} & ::= \text{domain of integers (including operations)} \\
\mathit{Bool} & ::= \text{domain of booleans (including operations)} \\
\mathit{AExp} ::= & \mathit{Int} \\
& \mid \mathit{Id} \\
& \mid \mathit{AExp} \texttt{ / } \mathit{AExp} \text{ [strict]} \\
& \mid \mathit{AExp} \texttt{ * } \mathit{AExp} \text{ [strict]} \\
& \mid \mathit{AExp} \texttt{ + } \mathit{AExp} \text{ [strict]} \\
& \mid \texttt{(} \mathit{AExp} \texttt{)} \\
\mathit{BExp} ::= & \mathit{Bool} \\
& \mid \mathit{AExp} \texttt{ <= } \mathit{AExp} \text{ [strict]} \\
& \mid \texttt{not } \mathit{BExp} \text{ [strict]} \\
& \mid \mathit{BExp} \texttt{ and } \mathit{BExp} \text{ [strict(1)]} \\
& \mid \texttt{(} \mathit{BExp} \texttt{)} \\
\mathit{Stmt} ::= & \texttt{skip} \\
& \mid \mathit{Id} \texttt{ = } \mathit{AExp} \\
& \mid \texttt{if } \mathit{BExp}\ \mathit{Stmt} \texttt{ else } \mathit{Stmt} \text{ [strict(1)]} \\
& \mid \texttt{\{} \mathit{Stmt} \texttt{\}} \\
& \mid \texttt{while } \mathit{BExp} \texttt{ do } \mathit{Stmt} \\
& \mid \mathit{Stmt} \texttt{ ; } \mathit{Stmt}
\end{array}
$$

Figure 4.1: $\mathbb{K}$ Syntax of IMP

4.1.1 IMP

Our running example is IMP, a simple imperative language intensively used in research papers. The $\mathbb{K}$ syntax of this programming language is shown in Figure 4.1. The IMP statements are either assignments, if statements, while loops, skip (i.e., the empty statement), or a list of statements.

The IMP configuration consists only of the program to be executed and an environment mapping variables to values. Precisely, the $\langle\rangle_{\mathsf{k}}$ cell contains the code and the $\langle\rangle_{\mathsf{env}}$ cell the environment:

$$\mathit{Cfg} \quad ::= \langle\langle \mathit{Code} \rangle_{\mathsf{k}} \langle \mathit{Map}_{\mathit{Id},\mathit{Int}} \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}}$$

The semantics of IMP is shown in Figure 4.2. As usual, $\mathbb{K}$ rules specify how the configuration evolves during the execution. Most syntactical constructs require only one semantical rule. The exceptions are the `and` boolean operation and the `if` statement, which have boolean arguments and require two rules each.

$$\begin{array}{l}
\langle\langle I_1 \texttt{ + } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 +_{\mathit{Int}} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle I_1 \texttt{ * } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 *_{\mathit{Int}} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle I_1 \texttt{ / } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 /_{\mathit{Int}} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} I_2 \neq_{\mathit{Int}} 0 \\
\langle\langle I_1 \texttt{ <= } I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle I_1 \leq_{\mathit{Int}} I_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \mathit{true} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \mathit{false} \texttt{ and } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{false} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \texttt{not } B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \neg B \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \texttt{skip } \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle S_1 \texttt{;} S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1 \curvearrowright S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \texttt{\{ } S \texttt{ \}} \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \texttt{if } \mathit{true} \ \ S_1 \texttt{ else } S_2 \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_1\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \texttt{if } \mathit{false} \ \ S_1 \texttt{ else } S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle S_2\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle \texttt{while } B \texttt{ do } S \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \Rightarrow \\
\qquad \langle\langle \texttt{if } B \texttt{ \{ } S \texttt{ ;while } B \texttt{ do } S \texttt{ \}else skip } \cdots\rangle_{\mathsf{k}} \cdots\rangle_{\mathsf{cfg}} \\
\langle\langle X \cdots\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \mathit{lookup}(X, M) \cdots\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \\
\langle\langle X \texttt{ = } I \cdots\rangle_{\mathsf{k}} \langle M\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow \langle\langle \cdots\rangle_{\mathsf{k}} \langle \mathit{update}(X, M, I)\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}
\end{array}$$

Figure 4.2: $\mathcal{S}_{\text{IMP}}$: the $\mathbb{K}$ Semantics of IMP

In addition to the rules shown in Figure 4.2 the semantics of IMP includes the heating/cooling rules, induced by the *strict* attribute. We show only the case of the `if` statement, which is strict in the first argument. The evaluation is done by the following rules:

$$\frac{}{A_1 + A_2 \Rightarrow A_1 \curvearrowright \Box + A_2} (heating) \qquad \frac{}{I_1 \curvearrowright \Box + A_2 \Rightarrow I_1 + A_2} (cooling)$$

$$\frac{}{A_1 + A_2 \Rightarrow A_2 \curvearrowright A_1 + \Box} \qquad \frac{}{I_2 \curvearrowright A_1 + \Box \Rightarrow A_1 + I_2}$$

Here, A_1 and A_2 are terms of sort $AExp$, I_1 and I_2 are integers, and $\Box$ is destined to receive an integer value (once it is computed, typically, by the other rules in the semantics).

Hoare Logic

Hoare Logic provides a formal system for reasoning about program correctness. Using the axioms and the deduction rules of the proof system one can prove Hoare triples $(\!|\psi|\!)St(\!|\psi'|\!)$, where St is a program statement, and ψ and ψ' are assertions involving program variables.

Definition 4.1.1 (Hoare triples) *A* Hoare triple *is a formula of the form $(\!|\psi|\!)St(\!|\psi'|\!)$, where St is a program statement, and ψ and ψ' are* assertions *defined as below:*

- *every boolean expression is an assertion;*
- *if ψ is an assertion then $\neg\psi$ is an assertion;*
- *if ψ and ψ' are assertions then $\psi \wedge \psi'$ is an assertion;*
- *if ψ is an assertion and V a set of variables then $(\exists V)\psi$ is an assertion,*

where the quantifier $\exists$, and the connectives $\neg$, $\wedge$ are the same as in FOL.

In IMP, boolean expressions correspond to sort *BExp*. The other known FOL connectives ($\rightarrow$, $\vee$) and the quantifier $\forall$ can be also used in assertions, being expressed in terms of the ones above as explained in Section 2.2.

Next, we define the satisfaction relation between a program configuration and an assertion:

$$[\textsf{HL-skip}]\frac{}{(\!|\psi|\!)\,\texttt{skip}\,(\!|\psi|\!)} \qquad [\textsf{HL-block}]\frac{(\!|\psi|\!)\,St\,(\!|\psi'|\!)}{(\!|\psi|\!)\,\{\ St\ \}\,(\!|\psi'|\!)}$$

$$[\textsf{HL-assign}]\frac{}{(\!|\psi|\!)\,\mathtt{a} = E\,(\!|(\exists u)\ \mathtt{a} = E[u/\mathtt{a}] \wedge \psi[u/\mathtt{a}]|\!)}$$

$$[\textsf{HL-seq}]\frac{(\!|\psi_1|\!)\,St_1\,(\!|\psi_2|\!) \quad (\!|\psi_2|\!)\,St_2\,(\!|\psi_3|\!)}{(\!|\psi_1|\!)\,St_1\ ;\ St_2\,(\!|\psi_3|\!)}$$

$$[\textsf{HL-while}]\frac{(\!|\psi \wedge BE|\!)\,St\,(\!|\psi|\!)}{(\!|\psi|\!)\,\texttt{while (}\ BE\ \texttt{) do}\ St\,(\!|\psi \wedge \neg BE|\!)}$$

$$[\textsf{HL-if}]\frac{(\!|\psi \wedge BE|\!)\,St_1\,(\!|\psi'|\!) \quad (\!|\psi \wedge \neg BE|\!)\,St_2\,(\!|\psi'|\!)}{(\!|\psi|\!)\,\texttt{if (}\ BE\ \texttt{)}\ St_1\ \texttt{else}\ St_2\,(\!|\psi'|\!)}$$

$$[\textsf{HL-cseq}]\frac{\psi \to \psi_1 \quad (\!|\psi_1|\!)\,St\,(\!|\psi'_1|\!) \quad \psi'_1 \to \psi'}{(\!|\psi|\!)\,St\,(\!|\psi'|\!)}$$

Figure 4.3: [HL-IMP]: Floyd-Hoare proof system for IMP

Definition 4.1.2 *Given a configuration* $\gamma \triangleq \langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *and an assertion* ψ *the* satisfaction *relation* $\gamma \models_{\text{HL}} \psi$*, read* γ *satisfies* ψ*, is defined as follows:*

- *if* ψ *is a boolean expression then* $\gamma \models_{\text{HL}} \psi$ *iff* $\langle\langle\psi\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow^*_{\mathcal{S}_{\text{IMP}}} \langle\langle true\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$*;*
- $\gamma \models_{\text{HL}} \neg\psi$ *iff* $\gamma \models_{\text{HL}} \psi$ *does not hold;*
- $\gamma \models_{\text{HL}} \psi \wedge \psi'$ *iff* $\gamma \models_{\text{HL}} \psi$ *and* $\gamma \models_{\text{HL}} \psi'$*;*
- $\gamma \models_{\text{HL}} (\exists V)\psi$ *iff there exists a configuration* $\tau \triangleq \langle\langle St\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *with* $\sigma'(x) = \sigma(x)$ *for all* $x \notin V$*, such that* $\tau \models_{\text{HL}} \psi$*.*

A triple $(\!|\psi|\!)\,St\,(\!|\psi'|\!)$ *is called* valid *(w.r.t. partial correctness) if and only if for any program configuration* $\gamma \triangleq \langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *such that* γ *satisfies* ψ*, if* $\langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow_{\mathcal{S}_{\text{IMP}}} \triangleq \langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *then* $\gamma' \triangleq \langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ *satisfies* ψ'*. We denote by* $\mathcal{S}_{\text{IMP}} \models_{\text{HL}} (\!|\psi|\!)\,St\,(\!|\psi'|\!)$ *the fact that* $(\!|\psi|\!)\,St\,(\!|\psi'|\!)$ *is valid for* $\mathcal{S}_{\text{IMP}}$*.*

The Hoare Logic deduction system for IMP is shown in Figure 4.3. It contains a set of deduction rules which correspond to each statement of the language. The deduction system can be used only to prove Hoare triples w.r.t partial correctness. In [100], the authors have presented and proved correct a technique to automatically translate Hoare triples into reachability rules and Hoare Logic proof derivations into Reachability Logic proof derivations. Based in this translation, the authors prove the soundness of the proof system shown in Figure 4.3 by relying on the soundness of Reachability Logic.

4.1.2 From Hoare Logic to Reachability Logic

For the IMP programming language, a Hoare triple $(\!|\psi|\!)St(\!|\psi'|\!)$ can be encoded as the following reachability rule [100]:

$$(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y} \qquad (4.1)$$

X and Y are two disjoint sets of variables corresponding to two sets `X`, `Y`, which contain the program variables which appear in St, and the additional variables that appear in ψ and ψ' but not in `X`, respectively `Y`. We use X and Y instead of `X` and `Y` because Reachability Logic distinguishes between program and logical variables. This restriction does not limit Hoare Logic in any way, because we can always choose `X` as being the set of all program variables about which we want to reason and `Y` the rest of variables occurring anywhere in the state or the specification, but different from `X`. The state σ_X maps each $\mathtt{x} \in \mathtt{X}$ to its corresponding $x \in X$. Formulas $\psi_{X,Y}$ and $\psi'_{X,Y}$ are ψ and respectively ψ' with $\mathtt{x} \in \mathtt{X}$ and $\mathtt{y} \in \mathtt{Y}$ replaced by their corresponding $x \in X$ and $y \in Y$, and each expression construct (e.g.+) replaced by its mathematical correspondent (e.g. $+_{Int}$). We use an additional function $eval_{\sigma_X}$ to perform the same transformation over program expressions (e.g. $eval_{\sigma_X}(\mathtt{x + y}) = x +_{Int} y$). If E is a program expression then $E' = eval_{\sigma_X}(E)$ is given by evaluating E in the environment σ_X, that is, $\langle\langle E\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \Rightarrow^* \langle\langle E'\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$, where E' is a symbolic expression of sort *Int* (and, therefore, $\langle\langle E'\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is irreducible).

Example 4.1.1 *Let* $(\!|\mathtt{a} > 0|\!)\mathtt{a} = \mathtt{a} + \mathtt{b}(\!|(\exists u)(\mathtt{a} = u + \mathtt{b} \wedge u > 0)|\!)$ *be a Hoare triple. The set of program variables is* $\mathtt{X} = \{\mathtt{a}, \mathtt{b}\}$*, while the set of logical*

variables is $X = \{a_X, b_X\}$. *Both* $\mathtt{Y}$ *and* Y *are empty sets. The state is* $\sigma_X = \{\mathtt{a} \mapsto a_X \mathtt{b} \mapsto b_X\}$, *and* $\psi_{X,Y} = a_X >_{Int} 0$ *and* $\psi'_{X,Y} = (\exists u)(a_X =_{Int} u +_{Int} b_X \wedge u >_{Int} 0)$. *By applying transformation (4.1) we obtain the following reachability rule:*

$$(\exists\{a_X, b_X\})\langle\langle \mathtt{a} = \mathtt{a} + \mathtt{b}\rangle_{\mathsf{k}}\langle \mathtt{a} \mapsto a_X\ \mathtt{b} \mapsto b_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow (\exists\{a_X, b_X\})\langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a} \mapsto a_X\ \mathtt{b} \mapsto b_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\exists u)(a_X =_{Int} u +_{Int} b_X \wedge u >_{Int} 0) \tag{4.2}$$

The assertion $\psi'_{X,Y}$ *was obtained by making use the* $eval_{\sigma_X}$ *function, as follows:* $eval_{\sigma_X}((\mathtt{a} + \mathtt{b})[u/\mathtt{a}]) = eval_{\sigma_X}(u + \mathtt{b}) = u +_{Int} b_X$.

We applied the transformation (4.1) to the proof system from Figure 4.3 and we obtained the equivalent proof system [HL'-IMP] given using reachability rules, which shown in Figure 4.4. This proof system is used in the next section to implement Hoare Logic using symbolic execution.

4.1.3 Hoare Logic by symbolic execution

The goal of this section is to show how to prove Hoare triples using symbolic execution. As pointed out previously in this section, triples $(\!|\psi|\!)St(\!|\psi|\!)$ can be proved by proving their corresponding reachability rules $\varphi \Rightarrow \varphi'$. Note that the transformation (4.1) generates a particular type of reachability rules, i.e. φ' contains an empty $\langle\cdot\rangle_{\mathsf{k}}$ cell. In the case of IMP, we can prove such a rule by executing symbolically φ, and then by verifying if every (result) pattern φ'' implies φ'. However, symbolic execution may not finish due to loops in programs, and thus, we need a way to provide invariants for loops.

In the following, we show how one can extend the definition of IMP with support not only for invariants, but also for preconditions and postconditions. This extension is done in two steps:

1. Add syntax and semantics for preconditions, postconditions, and invariants.
2. Define `assume` and `implies` operations for verifying RL formulas.

For IMP, the syntax is enriched with a new sort *Assert* for assertions, which extends sort *BExp* (boolean expressions of the language). This

$$[\mathsf{HL'\text{-}skip}]\ \frac{}{(\exists X)\langle\langle \texttt{skip}\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\rangle_{\mathsf{cfg}}}$$

$$[\mathsf{HL'\text{-}assign}]\ \frac{}{\begin{array}{l}(\exists X)\langle\langle \mathsf{a} = E\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow \\ \qquad (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge ((\exists u)(a_X = eval_{\sigma_X}(E[u/\mathsf{a}]) \wedge \psi_{X,Y}[u/eval_{\sigma_X}(\mathsf{a})]))\end{array}}$$

$$[\mathsf{HL'\text{-}seq}]\ \frac{\begin{array}{c}(\exists X)\langle\langle St_1\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^1_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^2_{X,Y} \\ (\exists X)\langle\langle St_2\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^2_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^3_{X,Y}\end{array}}{(\exists X)\langle\langle St_1 St_2\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^1_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^3_{X,Y}}$$

$$[\mathsf{HL'\text{-}block}]\ \frac{(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}}{(\exists X)\langle\langle \{\ St\ \}\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}}$$

$$[\mathsf{HL'\text{-}while}]\ \frac{(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge BE \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}}{\begin{array}{l}(\exists X)\langle\langle \texttt{while (}\ BE\ \texttt{) do}\ St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow \\ \qquad (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge \neg BE\end{array}}$$

$$[\mathsf{HL'\text{-}if}]\ \frac{\begin{array}{c}(\exists X)\langle\langle St_1\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge BE \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y} \\ (\exists X)\langle\langle St_2\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge \neg BE \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}\end{array}}{\begin{array}{l}(\exists X)\langle\langle \texttt{if (}\ BE\ \texttt{)}\ St_1\ \texttt{else}\ St_2\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow \\ \qquad (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}\end{array}}$$

$$[\mathsf{HL'\text{-}cseq}]\ \frac{\psi_{X,Y} \rightarrow \psi^1_{X,Y} \qquad \begin{array}{c}(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^1_{X,Y} \Rightarrow \\ (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi^2_{X,Y}\end{array} \qquad \psi_{X,Y}{}^2 \rightarrow \psi'_{X,Y}}{(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}}$$

Figure 4.4: [HL'-IMP]: the equivalent Floyd-Hoare proof system given using reachability rules. Each Hoare triple from the proof system in Figure 4.3 has been transformed into an equivalent reachability rule using (4.1).

allows boolean and arithmetic expressions (including program variables) in annotations. Alternatively, one can define a custom syntax for *Assert*, which also permits program variables. Once we have a sort for assertions we can give syntax for the preconditions, postconditions, and invariants. For IMP we assume that triples have the following syntax:

$$\texttt{pre:}\mathit{Assert}\ \texttt{post:}\ \mathit{Assert}\ \mathit{Stmt}$$

The syntax of the annotated `while` loop is:

$$\texttt{while}\ \mathit{BExp}\ \texttt{inv:}\ \mathit{Assert}\ \ \mathit{Stmt}$$

According to transformation (4.1), the precondition and the program become parts of the left hand side of the corresponding reachability rule. The right hand side of the reachability rule contains the postcondition, and an empty $\langle\cdot\rangle_{\mathsf{k}}$ cell. The semantics of a Hoare triple can be given using the additional constructs `assume` and `implies` (defined later), as follows:

$$\left\langle \frac{\texttt{pre:}\ \psi\ \texttt{post:}\ \psi'\ \mathit{St}}{\texttt{assume}\ (\langle \mathit{St}\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\langle\psi\rangle_{\mathsf{cond}})\curvearrowright\texttt{implies}\ (\langle\cdot\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\langle\psi'\rangle_{\mathsf{cond}})} \right\rangle_{\mathsf{k}}$$

The intuition of the rule above is that it assumes the code to be executed and the precondition, and after the execution, it checks if the postcondition holds. The $\langle\rangle_{\mathsf{cond}}$ cell is used to hold the path condition.

The operational semantics of the `while` loop with invariants, is given by the following rewrite rules:

$$\left\langle \frac{\texttt{while}(B)\texttt{inv:}\psi\ \mathit{St}\curvearrowright K}{\texttt{assume}(\langle \mathit{St}\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\langle B\wedge\psi\rangle_{\mathsf{cond}})\curvearrowright\texttt{implies}(\langle\cdot\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\langle\psi\rangle_{\mathsf{cond}})} \right\rangle_{\mathsf{k}}$$

$$\left\langle \frac{\texttt{while}(B)\texttt{inv:}\psi\ \mathit{St}\curvearrowright K}{\texttt{implies}(\langle\cdot\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\langle\psi\rangle_{\mathsf{cond}})\curvearrowright\texttt{assume}(\langle K\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\langle\neg B\wedge\psi\rangle_{\mathsf{cond}})} \right\rangle_{\mathsf{k}}$$

The above rules breaks the annotated `while` statement in two executions: one for proving the body of the loop: $\langle\langle \mathit{St}\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\wedge(B\wedge\psi)\Rightarrow\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\wedge\psi'$, and another one for proving if the invariant and the negation of the loop condition hold after the loop has been executed: $\langle\langle K\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\wedge(\neg B\wedge\psi)\Rightarrow\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}\wedge\psi'$.

The additional operations `assume` and `implies` are used to execute patterns and verify implications between patterns, respectively. The `assume`(φ) construct, prepares the pattern given as argument, i.e. φ, for symbolic execution by setting up the initial configuration. In the case of IMP, the semantics of `assume` is the following:

$$\langle \underbrace{\texttt{assume}\ (\langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\langle\phi\rangle_{\mathsf{cond}})\rangle_{\mathsf{cfg}}}_{St} \cdots\rangle_{\mathsf{k}}\langle \underbrace{\_}_{\sigma} \rangle_{\mathsf{env}}\langle \underbrace{\_}_{\phi} \rangle_{\mathsf{cond}}$$

The $\mathbb{K}$ tool produces by symbolic execution of φ one or more result configurations (patterns). For any such result, say φ'', we have to check whether $\varphi'' \rightarrow \varphi'$. This is done by using the `implies`(φ') operation, which checks if the current symbolic configuration φ'', implies the pattern given as argument φ'. The following proposition allows us to reduce the verification of $\varphi'' \rightarrow \varphi'$ to a formula which can be sent to an SMT solver. We denote by $\sigma(\texttt{x})$ the value which is mapped to program variable `x` by σ, and by $range(\sigma)$ the set $\{\sigma(\texttt{x}) \mid \texttt{x} \in \texttt{X}\}$, where `X` is the set of program variables.

Proposition 4.1.1 *Any pattern $\varphi \triangleq \pi \wedge \phi$, where $\pi \triangleq \langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}$ is in relation $\sim$ (cf. Definition 3.3.1) with the pattern $(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)$, where $\bigwedge_\sigma \triangleq \bigwedge_{x\in X} x = \sigma(\texttt{x})$.*

Proof From Lemma 2.3.1 we have $[\![(\exists X)\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)]\!] = [\![\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)]\!]$. Therefore, we have it is sufficient to show that $[\![\varphi]\!] = [\![\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)]\!]$.

($\subseteq$) Let $\gamma \in [\![\varphi]\!]$. Then, there is a valuation $\rho : Var \rightarrow \mathcal{T}$ such that $(\gamma, \rho) \models_{\text{ML}} \pi \wedge \phi$, that is, $\rho(\pi) = \gamma$ and $\rho \models_{\text{ML}} \phi$. We consider a substitution $f_\sigma : X \rightarrow range(\sigma)$ which is defined as $f_\sigma(x) = \sigma(\texttt{x})$, for all $x \in X$. Next, we consider a valuation $\rho' : Var \rightarrow \mathcal{T}$ such that $\rho' = \rho \circ f_\sigma$. Then, we have $\rho'(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}) = (\rho \circ f_\sigma)(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}) = \rho(f_\sigma(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}})) = \rho(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}) = \rho(\pi) = \gamma$, and $\rho'(\phi) = (\rho \circ f_\sigma)(\phi) = \rho(f_\sigma(\phi)) = \rho(\phi) = true$, since ϕ does not contain variables from X. Moreover, we have $\rho'(\bigwedge_\sigma) = (\rho \circ f_\sigma)(\bigwedge_\sigma) = \rho(f_\sigma(\bigwedge_{x\in X} x = \sigma(\texttt{x}))) = \rho(\bigwedge_{x\in X} f_\sigma(x) = f_\sigma(\sigma(\texttt{x}))) = \rho(\bigwedge_{x\in X} \sigma(\texttt{x}) = \sigma(\texttt{x})) = true$. So, we have ρ' such that $(\gamma, \rho') \models_{\text{ML}} \langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)$

($\supseteq$) Let $\gamma \in [\![\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)]\!]$. Then, there is a valuation $\rho' : Var \rightarrow \mathcal{T}$, such that $(\gamma, \rho') \models_{\text{ML}} \langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi \wedge \bigwedge_\sigma)$.

Let $\rho : Var \to \mathcal{T}$ be a valuation, such that $\rho' = \rho \circ f_\sigma$. Then, we obtain $\gamma = \rho'(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}) = (\rho \circ f_\sigma)(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}) = \rho(f_\sigma(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}})) = \rho(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}}) = \rho(\pi)$, and from $\rho'(\phi) = (\rho \circ f_\sigma)(\phi) = \rho(f_\sigma(\phi)) = \rho(\phi)$ we obtain $\rho \models_{\mathrm{ML}} \phi$. Therefore, there is ρ such that $(\gamma, \rho) \models_{\mathrm{ML}} \pi \wedge \phi$, i.e. $\gamma \in [\![\varphi]\!]$.
■

Example 4.1.2 *If* $\varphi \triangleq \langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto 4\ \mathtt{n} \mapsto t+1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge true$ *is a pattern then, by Proposition 4.1.1,* $\varphi \sim (\exists\{x, n\})\langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{x} \mapsto x\ \mathtt{n} \mapsto n\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge true \wedge (x = 4 \wedge n = t+1)$.

The implication $\varphi'' \to \varphi'$ can be rewritten as $\langle\langle St\rangle_{\mathsf{k}}\langle\sigma''\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \phi'' \to \langle\langle St\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \phi'$. Next, we use Proposition 4.1.1, an we simplify the implication using the following equivalences:

$(\exists X'')\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_{X''}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi'' \wedge \bigwedge_{\sigma''}) \to (\exists X')\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_{X'}\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi' \wedge \bigwedge_{\sigma'})$ iff

$(\exists X)(\langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi'' \wedge \bigwedge_{\sigma''}) \to \langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge (\phi' \wedge \bigwedge_{\sigma'}))$ iff

$(\exists X)((\phi'' \wedge \bigwedge_{\sigma''}) \to (\phi' \wedge \bigwedge_{\sigma'}))$ iff $(\exists X)(\neg(\phi'' \wedge \bigwedge_{\sigma''}) \vee (\phi' \wedge \bigwedge_{\sigma'}))$ iff

$((\phi'' \wedge \bigwedge_{\sigma''}) \wedge \neg(\phi' \wedge \bigwedge_{\sigma'}))$ is not satisfiable.

We always assume that φ'' and φ' contain an empty $\langle\cdot\rangle_{\mathsf{k}}$ cell. Now, we give semantics to `implies`:

$$\langle \underline{\mathtt{implies}\ (\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma'\rangle_{\mathsf{env}}\langle\psi'\rangle_{\mathsf{cond}}\rangle_{\mathsf{cfg}}\rangle_{\mathsf{k}}}\langle\sigma''\rangle_{\mathsf{env}}\langle\phi''\rangle_{\mathsf{cond}}$$

$$\mathtt{requires}\ unsat(\phi'' \wedge \bigwedge_{\sigma''} \wedge \neg(\phi' \wedge \bigwedge_{\sigma'}))$$

In practice, one may need additional constructs for defining $\bigwedge_\sigma$. For IMP, we have defined a function which traverses the $\langle\rangle_{\mathsf{env}}$ cell and computes $\bigwedge_\sigma$.

In the rest of the section, we let $\mathcal{S}^{\mathbf{5}}_{\mathrm{IMP}'}$ be $\mathcal{S}^{\mathbf{5}}_{\mathrm{IMP}}$ enriched with the rules shown above. Below we define irreducible patterns and successful executions.

Definition 4.1.3 (irreducible pattern) *Let φ and φ' be patterns. φ is* irreducible *iff there is no φ' such that $\varphi \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} \varphi'$.*

Definition 4.1.4 (complete and successful execution) *A* complete execution *is an execution $\varphi \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} \ldots \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} \varphi'$ such that φ' is irreducible. A complete execution is* successful*, and we write $\varphi \Rightarrow^! \varphi'$, if φ' contains an empty $\langle\cdot\rangle_{\mathsf{k}}$ cell. We say that $\varphi\!\downarrow$ holds iff all complete executions starting from φ are successful.*

Example 4.1.3 *Let $\varphi \triangleq \langle\langle \mathsf{a} = \mathsf{a} + 1\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0$ (the left hand side of rule (4.2)). Symbolic execution of φ is shown below:*

$$
\begin{array}{ll}
\varphi \triangleq \langle\langle \mathsf{a} = \mathsf{a} + 1\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(heating)} \\
\langle\langle \mathsf{a} + 1 \curvearrowright \mathsf{a} = \square\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(heating)} \\
\langle\langle \mathsf{a} \curvearrowright \square + 1 \curvearrowright \mathsf{a} = \square\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(lookup)} \\
\langle\langle a_X \curvearrowright \square + 1 \curvearrowright \mathsf{a} = \square\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(cooling)} \\
\langle\langle a_X + 1 \curvearrowright \mathsf{a} = \square\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(addition)} \\
\langle\langle a_X +_{Int} 1 \curvearrowright \mathsf{a} = \square\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(cooling)} \\
\langle\langle \mathsf{a} = a_X +_{Int} 1\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \Rightarrow_{\mathcal{S}^s_{\text{IMP}'}} & \textit{(update)} \\
\langle\langle\cdot\rangle_{\mathsf{k}}\langle \mathsf{a} \mapsto a_X +_{Int} 1\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge a_X >_{Int} 0 \triangleq \varphi' &
\end{array}
$$

Note that this execution is successful, i.e. $\varphi \Rightarrow^! \varphi'$ and $\varphi\!\downarrow$.

Next, we define the set of reachability rules derived from φ as follows:

Definition 4.1.5 *Let φ be a pattern. We define the set $RL(\varphi)$ of reachability rules as follows:*

$$RL(\varphi) = \{\varphi \Rightarrow \varphi' \mid \varphi \Rightarrow^! \varphi'\}$$

Example 4.1.4 *Recall φ and φ' from Example 4.1.3. Then, $RL(\varphi) = \{\varphi \Rightarrow \varphi'\}$.*

$RL(\varphi)$ contains all the reachability rules $\varphi \Rightarrow \varphi'$ given by successful executions $\varphi \Rightarrow^! \varphi'$.

Theorem 4.1.1 *Let $\varphi \triangleq \langle\langle St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}$ be a pattern. Then any reachability rule $\varphi \Rightarrow \varphi' \in RL(\varphi)$ can be proved using* [*HL'*-IMP].

Proof We proceed by induction on St:

- $St \triangleq \texttt{skip}$. In this case, symbolic execution of $\langle\langle \texttt{skip} \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}$ consists in applying directly the rule for `skip` from $\mathcal{S}^{\mathfrak{s}}_{\mathrm{IMP}'}$, and we obtain $\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}$. Thus, symbolic execution derives $(\exists X)\langle\langle \texttt{skip} \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}$, which corresponds to [HL'-skip].

- $St \triangleq \mathtt{a} = E$. After the symbolic execution of $\langle\langle \mathtt{a} = E \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}$, the variable `a` is mapped in the new environment to the (evaluated) expression $eval_{\sigma_X}(E)$. Moreover, $\psi_{X,Y}$ holds for the previous value of the program variable `a`, that is $eval_{\sigma_X}(\mathtt{a})$. Therefore, by symbolic execution we derive $(\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge ((\exists u)(a_X = eval_{\sigma_X}(E[u/\mathtt{a}]) \wedge \psi_{X,Y}[u/eval_{\sigma_X}(\mathtt{a})]))$, which corresponds to [HL'-assign].

- $St \triangleq \{\ S\ \}$. From the inductive hypothesis, symbolic execution derives $(\exists X)\langle\langle S \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$. The symbolic execution of $\{\ S\ \}$, consists in applying first the rule for blocks, which reduces the execution of the block to the execution of the body. Using the inductive hypothesis, we obtain $(\exists X)\langle\langle \{\ St\ \} \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$. In the proof system, this corresponds to [HL'-block].

- $St \triangleq St_1\ \mathtt{;}\ St_2$. From the inductive hypothesis, symbolic execution derives $(\exists X)\langle\langle St_1 \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$, and $(\exists X)\langle\langle St_2 \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$ in [HL'-IMP]. Symbolic execution of a sequence consists in executing St_1 followed by St_2. Since we have the proofs for St_1 and St_2, we obtain $(\exists X)\langle\langle St_1\ St_2 \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$. The coresponding proof rule is [HL'-seq].

- $St \triangleq \texttt{while (}\ BE\ \texttt{)}\ S$. Let $\psi_{X,Y}$ be an invariant for this loop. In this case, symbolic execution consists in analysing two possible paths: either BE is *true* and the execution continues by assuming $\langle\langle S \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge BE$, or BE is *false* and the rule for `implies` checks whether the path condition implies the invariant and $\neg BE$. First, symbolic execution derives (from the inductive hypothesis) $(\exists X)\langle\langle S \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge BE \Rightarrow (\exists X)\langle\langle \cdot \rangle_{\mathsf{k}} \langle \sigma_X \rangle_{\mathsf{env}} \rangle_{\mathsf{cfg}} \wedge \psi_{X,Y}$.

Second, it derives $(\exists X)\langle\langle \texttt{while (} BE \texttt{) do } St\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge \neg BE$ by verifying the implication. This corresponds to [HL'-while] in the proof system.

- $St \triangleq \texttt{if (} B \texttt{) } St_1 \texttt{ else } St_2$. Symbolic execution of the `if` statement consists in exploring the two possible branches, St_1 and St_2, adding to the path condition the corresponding constraint, i.e. $\psi_{X,Y} \wedge B$ and $\psi_{X,Y} \wedge \neg B$, respectively. From the induction hypothesis we have $(\exists X)\langle\langle St_1\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge B \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$ and $(\exists X)\langle\langle St_2\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \wedge \neg B \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$. By applying the corresponding rules in the semantics and the inductive hypothesis, we obtain $(\exists X)\langle\langle \texttt{if (} B \texttt{) } St_1 \texttt{ else } St_2\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi_{X,Y} \Rightarrow (\exists X)\langle\langle\cdot\rangle_{\mathsf{k}}\langle\sigma_X\rangle_{\mathsf{env}}\rangle_{\mathsf{cfg}} \wedge \psi'_{X,Y}$. This corresponds to [HL'-if] proof rule.

■

Essentially, Theorem 4.1.1 states that any reachability rule $\varphi \Rightarrow \varphi' \in RL(\varphi)$ obtained by (successful) symbolic execution of φ can be derived using the [HL'-IMP] proof system. Note that, in this approach, we verify a particular type of reachability rules, i.e. those which are generated from Hoare triples. For such rules, we only need symbolic execution and an SMT solver for verification.

The way we apply the proof rules from [HL'-IMP] proof system can be seen as a strategy for applying the Reachability Logic proof system on a particular language. In Section 4.2 we present a more general approach for proving RL formulas, which is based on symbolic execution and also handles circular behaviours in programs.

4.2 Reachability-Logic Verification

As presented in Section 2.4, Reachability Logic (RL) is a language independent logic for specifying program properties. For instance, on the `gcd` program shown in Fig. 4.5, the RL formula

$\langle \texttt{gcd}\rangle_{\mathsf{k}}\langle \texttt{a}\mapsto l_{\texttt{a}}\ \texttt{b}\mapsto l_{\texttt{b}}\ \texttt{x}\mapsto l_{\texttt{x}}\ \texttt{y}\mapsto l_{\texttt{y}}\ \texttt{r}\mapsto l_{\texttt{r}}\rangle_{\mathsf{env}}$

```
x = a;  y = b;
while (y > 0){
  r = x % y;
  x = y;
  y = r;
}
```

Figure 4.5: CinK program:`gcd`

$$\langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x\, l_{\mathtt{y}} \mapsto y\, l_{\mathtt{r}} \mapsto r \rangle_{\mathsf{store}}\ \sigma \wedge a \geq 0 \wedge b \geq 0 \Rightarrow$$
$$(\exists x')(\exists y')(\exists r')\langle \cdot \rangle_{\mathsf{k}} \langle \mathtt{a} \mapsto l_{\mathtt{a}}\ \mathtt{b} \mapsto l_{\mathtt{b}}\ \mathtt{x} \mapsto l_{\mathtt{x}}\ \mathtt{y} \mapsto l_{\mathtt{y}}\ \mathtt{r} \mapsto l_{\mathtt{r}} \rangle_{\mathsf{env}}$$
$$\langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x'\, l_{\mathtt{y}} \mapsto y'\, l_{\mathtt{r}} \mapsto r' \rangle_{\mathsf{store}}\ \sigma \wedge x' = gcd(a, b), \tag{4.3}$$

where $\sigma \triangleq \langle R \rangle_{\mathsf{return}} \langle S \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}}$, specifies that after the complete execution of the `gcd` program from a configuration where the program variables `a`, `b` are bound to non-negative values a, b, a configuration where the variable `x` is bound to a location which holds the value $gcd(a, b)$ is reached. Here, *gcd* is a mathematical definition of the greatest-common-divisor ($gcd(0, 0) = 0$ by convention), and *lookup* is function is a standard lookup function in associative maps.

A naive attempt at verifying the RL formula (4.3) consists in symbolically executing the semantics of the CinK language with the `gcd` progam in its left-hand side, i.e., running `gcd` with symbolic values $a, b \geq 0$ for `a`,`b`, and searching for a configuration matched by the formula's right-hand side. However, this does not succeed because it gets caught into an infinite symbolic execution, induced by the infinitely many iterations of the loop.

The proof system of RL (Figure 2.1) is compact and elegant but, despite its nice theoretical properties, its use in practice on nontrivial programs is difficult, because it gives the user a lot of freedom regarding the order and manner of rule application, and offers no practical guidelines for constructing proofs. Moreover, it is not designed for disproving formulas: since the system is relatively complete, the only way to disprove a formula is to show that there exists no proof-tree for the formula (in the presence of an oracle able to decide the validity of first-order assertions), which is not practically possible. On the other hand, the proof system of RL is language independent, unlike in Hoare Logic, where the

proof system depends explicitly on the language. By its nature, verification based on Hoare Logic is a bit limited. First, it is not easy to express with Hoare triples properties for programs which involve recursive function calls, blocks with local variables, virtual method calls, etc. Second, since Hoare Logic is language dependent, its soundness has to proved for every language, which is not the case of Reachability Logic.

In this section we show how symbolic execution can be used for verifying RL formulas. The approach consists in a simpler proof system, where symbolic execution is a main ingredient and is used as a rule during proof construction. We show that a certain strategy for executing the proof system is sound (when it terminates successfully on a given input - set of RL formulas - then the formulas are valid) and is also weakly complete (if it terminates in failure then its input is invalid). The strategy is not relatively complete, since it may not terminate even for valid RL formulas, as illustrated by the naive symbolic execution attempt at proving (4.3). In order to terminate, it requires additional information under the form of RL formulas. Together, these properties say that when it terminates, our approach correctly solves the RL-based program-verification problem. Termination, of course, cannot be guaranteed because the RL verification problem is undecidable, but the soundness and weak completeness results say that any inability to prove/disprove formulas is only due to issues inherent to the RL verification problem, and not to the particular approach we propose for solving it.

The soundness and weak completeness results are based on certain mutual-simulation properties relating symbolic and concrete program execution, and on a so-called *circularity principle* for reachability-logic formulas, which specifies the conditions under which goals can be reused as hypotheses in proofs. This is essential for proving programs with infinite state-spaces induced e.g., by an unbounded (symbolic) number of loop iterations or of recursive calls. Soundness also requires that the semantics of the programming language is *total*; the behaviour of instructions is completely specified, and weak completeness additionally requires that the semantics is *confluent*: any two executions of a program eventually reach the same state. Weak completeness also poses certain additional requirements on the RL goals; none of these requirements is hard to meet.

In order to prove that a semantics $\mathcal{S}$ satisfies a set G of RL formulas called *goals*, i.e., for proving $\mathcal{S} \models_{\mathrm{RL}} G$ (i.e., $\mathcal{S} \models_{\mathrm{RL}} g$ for all $g \in G$) one can reuse the goals G as hypotheses, provided that the set of goals to be proved is replaced by another set of goals $\Delta_{\mathcal{S}}(G)$, obtained by symbolic execution from the set G. This goal-as-hypothesis reuse is essential for proving programs with unbounded execution lengths induced by loops of recursive function calls. Thus, from $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ one deduces $\mathcal{S} \models_{\mathrm{RL}} G$; we call this implication the *circularity principle* for RL. Here, $\Vdash$ denotes the entailment of a certain proof system, presented below, in which symbolic execution also plays a major role.

The following definition of *derivative* is an essential operation in RL verification approach. It computes (up to the equivalence relation $\sim$) a disjunction of all the successors of a pattern by the symbolic transition relation. It uses the choice operation ε, which chooses an arbitrary element in a nonempty set.

Definition 4.2.1 (Derivative) *The* derivative $\Delta_{\mathcal{S}}(\varphi)$ *of a pattern* φ *for a set* $\mathcal{S}$ *of rules is* $\Delta_{\mathcal{S}}(\varphi) \triangleq \bigvee_{[\varphi]_\sim \Rightarrow^{\mathrm{s}}_{\mathcal{S}} [\varphi']_\sim} \varepsilon([\varphi']_\sim)$.

Remark 4.2.1 *Since the symbolic transition relation is finitely branching (Remark 3.3.6), for finite rule sets* $\mathcal{S}$ *the derivative is a finite disjunction. Note also that the patterns in the derivative are only defined up to the equivalence relation* $\sim$.

Definition 4.2.2 (Derivable Pattern) *An elementary pattern* $\varphi \triangleq \pi \wedge \phi$ *is* derivable *for* $\mathcal{S}$ *if* $\Delta_{\mathcal{S}}(\varphi)$ *is a nonempty disjunction.*

The notion of cover, defined below, is essential for the soundness of RL-formula verification by symbolic execution, in particular, in situations where a proof goal is circularly used as a hypothesis. Such goals can only be used in symbolic execution only when they *cover* the pattern being symbolically executed:

Definition 4.2.3 (Cover) *Consider an elementary pattern* $\varphi \triangleq \pi \wedge \phi$. *A set of rules* $\mathcal{S}'$ *satisfying* $\models_{\mathrm{FOL}} \phi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} \sigma^{\pi_1}_{\pi}(\phi_1 \wedge \phi_2)$, *where* $\sigma^{\pi_1}_{\pi}$ *is the most general unifier of* π *and* π_1, *is a* cover *of* φ.

Remark 4.2.2 *The existence of the most-general unifier $\sigma_\pi^{\pi_1}$ in the above definition means the basic patterns in the LHS of rules in $\mathcal{S}'$ are unifiable with the basic pattern π. In particular, $\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2)$ is a nonempty disjunction, otherwise the validity in Def. 4.2.3 would not hold (an empty disjunction is equivalent to false).*

The notion of *total* semantics is essential for the soundness of our approach.

Definition 4.2.4 (Total Semantics) *We say that a set $\mathcal{S}$ of semantical rules is* total *if for all patterns φ derivable for $\mathcal{S}$, $\mathcal{S}$ is a cover for φ.*

Remark 4.2.3 *The semantics of CinK is not total because the rule for division:*

$$\langle I_1 \swarrow I_2 \curvearrowright C \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}} \wedge I_2 \neq_{Int} 0 \Rightarrow \\ \langle I_1 /_{Int} I_2 \curvearrowright C \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle St \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle S \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}}$$

does not meet the condition of Definition 4.2.4, since the semantics of CinK does not cover the patterns of the form:

$$\langle I_1 \swarrow I_2 \curvearrowright C \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}}$$

which are derivable. This is due to the condition $I_2 \neq 0$, which is not logically valid. The semantics can easily be made total by adding a rule

$$\langle I_1 \swarrow I_2 \curvearrowright C \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}} \wedge I_2 =_{Int} 0 \Rightarrow \\ \langle error \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}} \tag{4.4}$$

that leads divisions by zero into "error" configurations. We assume hereafter that the CinK semantics has been transformed into a total one by adding the above rule.

Using the notion of cover we obtain a derived rule of the RL proof system:

Lemma 4.2.1 *If $\mathcal{S}' \subseteq \mathcal{S}$ is a cover for φ, and G is a (possibly empty) set of RL formulas, then $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$.*

Proof Let $\varphi \triangleq \pi \wedge \phi$. According to Definition 4.2.1, we have $\Delta_{\mathcal{S}'}(\varphi) \triangleq \bigvee_{[\varphi]_\sim \Rightarrow^{\mathsf{s}}_{\mathcal{S}'} [\varphi']_\sim} \varepsilon([\varphi']_\sim)$. Since $\mathcal{S}'$ is a cover for φ, by Remark 4.2.2, $\Delta_{\mathcal{S}'}(\varphi)$ is a nonempty disjunction. Using Definition 3.3.2 we obtain that $[\varphi']_\sim = [\sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2)]_\sim$ for some $\alpha \triangleq (\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in S$ and $\sigma_\pi^{\pi_1}$ the most general symbolic unifier of π, π_1 as built in the proof of Lemma 3.2.1. The projection of $\sigma_\pi^{\pi_1}$ on $vars(\pi)$ is the identity, and the projection on $vars(\pi_1)$ is a substitution of $vars(\pi_1)$ matching π_1 on π. By a variable renaming we can always assume that $vars(\phi) \cap vars(\pi_1) = \emptyset$, which means that the effect of $\sigma_\pi^{\pi_1}$ on ϕ is the identity as well, i.e., $\sigma_\pi^{\pi_1}(\phi) = \phi$.

Using the above characterisation for the patterns φ', we obtain that the choices of the ε operation in $\Delta_{\mathcal{S}'} \triangleq \bigvee_{[\varphi]_\sim \Rightarrow^{\mathsf{s}}_{\mathcal{S}'} [\varphi']_\sim} \varepsilon([\varphi']_\sim)$ can be made such that

$$\Delta_{\mathcal{S}'}(\varphi) = \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} \sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi \wedge \phi_1 \wedge \phi_2) \tag{4.5}$$

On the other hand, by using the RL proof system from Figure 2.1 and the derived rules of RL (shown in Section 2.4):

- first apply *Substitution* with the rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2$ and substitution $\sigma_\pi^{\pi_1}$:

  $\mathcal{S} \vdash_G \sigma_\pi^{\pi_1}(\pi_1 \wedge \phi_1) \Rightarrow \sigma_\pi^{\pi_1}(\pi_2 \wedge \phi_2)$

- apply *LogicalFraming* with the patternless formula $\sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_2)$:

  $\mathcal{S} \vdash_G \sigma_\pi^{\pi_1}(\pi_1 \wedge \phi_1) \wedge \sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_2) \Rightarrow$
  $\sigma_\pi^{\pi_1}(\pi_2 \wedge \phi_2) \wedge \sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_2)$

- using the [Consequence] and the fact that FOL patternless formulas distribute over patterns:

  $\mathcal{S} \vdash_G \sigma_\pi^{\pi_1}(\pi) \wedge (\sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2)) \Rightarrow$
  $\sigma_\pi^{\pi_1}(\pi_2) \wedge (\sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2))$

  Since the effect of $\sigma_\pi^{\pi_1}$ on both π and ϕ is the identity, we further obtain:

  $\mathcal{S} \vdash_G \pi \wedge (\phi \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2)) \Rightarrow$
  $\sigma_\pi^{\pi_1}(\pi_2) \wedge (\sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2))$

- then, using CaseAnalysis and Consequence several times we obtain:

$$\mathcal{S} \vdash_G \pi \wedge \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} (\phi \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2)) \Rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}'} (\sigma_\pi^{\pi_1}(\pi_2) \wedge (\sigma_\pi^{\pi_1}(\phi) \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2))) \tag{4.6}$$

We know from (4.5) that the right-hand side of (4.6) is $\Delta_{\mathcal{S}'}(\varphi)$. To prove $\mathcal{S} \vdash_G \varphi \Rightarrow \Delta_{\mathcal{S}'}(\varphi)$ there only remains to prove ($\diamondsuit$): the condition in the left-hand side: $\bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\phi \wedge \sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2))$ is logically implied by ϕ in FOL. Since $\mathcal{S}'$ is a cover for φ, we obtain, using Definition 4.2.3, the validity of $\phi \rightarrow \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}'} (\sigma_\pi^{\pi_1}(\phi_1) \wedge \sigma_\pi^{\pi_1}(\phi_2))$, which proves ($\diamondsuit$) and concludes the proof of the lemma.

■

Corollary 4.2.1 *If $\mathcal{S}$ is total and φ is derivable for $\mathcal{S}$, then $\mathcal{S} \vdash \varphi \Rightarrow \Delta_{\mathcal{S}}(\varphi)$.*

Proof Apply Lemma 4.2.1 with an empty set G. ■

Before we introduce our proof system we need to deal with the issue that operational semantics are not always weakly well-defined as required by the RL original deductive system's soundness. For example, the semantics of CinK is not weakly well-defined due to the rules for division and modulo, which, for valuations ρ mapping divisors to 0, have no instance of their right-hand side. However, due to the introduction of the rule (4.4) in order to make the semantics total (cf. Remark 4.2.3), the semantics of division can now be equivalently rewritten using just one (reachability-logic) disjunctive rule:

$$\begin{array}{l} \langle I_1 \ / \ I_2 \curvearrowright C \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}} \Rightarrow \\ \quad (\langle I_1 \,/_{Int}\, I_2 \curvearrowright C \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}} \wedge I_2 \neq_{Int} 0) \\ \quad \vee (\langle error \rangle_{\mathsf{k}} \langle E \rangle_{\mathsf{env}} \langle S \rangle_{\mathsf{store}} \langle R \rangle_{\mathsf{return}} \langle St \rangle_{\mathsf{stack}} \langle I \rangle_{\mathsf{in}} \langle O \rangle_{\mathsf{out}} \wedge I_2 =_{Int} 0) \end{array} \tag{4.7}$$

By using this rule instead of the two original ones, and by applying the same transformation for the rules defining modulo, the semantics becomes weakly well-defined. This transformation is formalised as follows.

Definition 4.2.5 ($\mathcal{S}^\Delta$) *Given a set of semantical rules $\mathcal{S}$, the set of semantical rules $\mathcal{S}^\Delta$ is defined by $\mathcal{S}^\Delta \triangleq \{\pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \mid (\pi \wedge \phi \Rightarrow \varphi) \in \mathcal{S}\}$.*

The following lemma establishes that $\mathcal{S}^\Delta$ has both properties (totality and weak well definedness) required for the soundness of our approach. For this, we need to extend the notion of derivative for rules of the form $\mathcal{S}^\Delta$ (containing disjunctions in their right-hand side) by letting $\Delta_{\mathcal{S}^\Delta}(\varphi) \triangleq \Delta(\varphi)$. The derivability of a pattern for $\mathcal{S}^\Delta$ is also, by definition, the derivability for $\mathcal{S}$. The notion of cover is extended for such rules, of the form $\pi_1 \wedge \phi_1 \Rightarrow \bigvee_{j\in J} \pi_2^j \wedge \phi_2^j$ by letting $\varphi \triangleq \pi \wedge \phi$ be covered by a set $\mathcal{S}'$ of such rules if $\models_{\text{ML}} \varphi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \bigvee_{j\in J} \pi_2^j \wedge \phi_2^j \in \mathcal{S}'} \bigvee_{j\in J} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_j^2)$.

Lemma 4.2.2 *If $\mathcal{S}$ is total then $\mathcal{S}^\Delta$ is total and weakly well-defined.*

Proof We start by proving the *totality of $\mathcal{S}^\Delta$.* If $\mathcal{S}$ is total then, by definition, $\mathcal{S}$ covers all patterns φ derivable for $\mathcal{S}$. We need to prove that for all φ that is derivable for $\mathcal{S}^\Delta$, $\mathcal{S}^\Delta$ is a cover for φ. Assume such a $\varphi \triangleq \pi \wedge \phi$ derivable for $\mathcal{S}^\Delta$. We have defined derivability for $\mathcal{S}^\Delta$ as derivability for $\mathcal{S}$, hence φ is derivable for $\mathcal{S}$. The totality of $\mathcal{S}$ and the definition of cover then ensures $\models_{\text{FOL}} \phi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2)$(♣).

Now, the rules in $\mathcal{S}^\Delta$ are of the form $\pi \Rightarrow \Delta_{\mathcal{S}}(\pi)$, for some π occurring in the pattern LHS of a rule in $\mathcal{S}$. Up to FOL conditions we have $\Delta_{\mathcal{S}}(\pi) = \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\pi_2 \wedge \phi_1 \wedge \phi_2)$ (implicitly, the disjunction performed over those rule in $\mathcal{S}$ for which $\sigma_\pi^{\pi_1}$ exists). Hence, by the notion of cover applied to $\mathcal{S}^\Delta$, in order to show that $\mathcal{S}^\Delta$ is total, we need to show

$$\models_{\text{FOL}} \phi \rightarrow \bigvee_{\left(\pi \Rightarrow \bigvee_{(\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2) \in \mathcal{S}} \sigma_\pi^{\pi_1}(\pi_2 \wedge (\phi_1 \wedge \phi_2))\right) \in \mathcal{S}^\Delta} \left(\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2) \right) \tag{4.8}$$

where the second disjunction $\bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2)$ is merely repeated by the first disjunction $\bigvee_{(\ldots)\in\mathcal{S}^\Delta}$ as many times as rules in $\mathcal{S}^\Delta$. Hence, (4.8) simplifies to $\models_{\text{FOL}} \phi \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2)$, which we have obtained above (♣).

For *weak well-definedness of* $\mathcal{S}^\Delta$ we need to show that for $\pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \in \mathcal{S}^\Delta$ and each valuation ρ, there exists a configuration γ such that $(\gamma, \rho) \models_{\text{ML}}$

$\Delta_S(\pi)$. We have obtained above that $\Delta_{\mathcal{S}}(\pi) = \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\pi_2 \wedge (\phi_1 \wedge \phi_2))$ (where implicitly the disjunction is performed over those rules in $\mathcal{S}$ for which $\sigma_\pi^{\pi_1}$ exists). Now, $\mathcal{S}$ is total, and $\pi (= \pi \wedge \mathit{true})$ is derivable for $\mathcal{S}$. This implies $\models_{\text{FOL}} \mathit{true} \rightarrow \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2)$, i.e., $\models_{\text{FOL}} \bigvee_{\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}} \sigma_\pi^{\pi_1}(\phi_1 \wedge \phi_2)$. This means that for any ρ there exists in the above disjunction at least one disjunct, say, $\sigma_\pi^{\pi_1}(\phi_1^i \wedge \phi_2^i)$, such that $\rho \models_{\text{FOL}} \sigma_\pi^{\pi_1}(\phi_1^i \wedge \phi_2^i)$. By taking the corresponding $\gamma \triangleq \rho(\sigma_\pi^{\pi_1}(\pi_2))$ we obtain $(\gamma, \rho) \models_{\text{ML}} \sigma_\pi^{\pi_1}(\pi_2) \wedge \sigma_\pi^{\pi_1}(\phi_1^i \wedge \phi_2^i)$, i.e., (γ, ρ) models one of the disjuncts of $\Delta_{\mathcal{S}}(\pi)$. This implies $(\gamma, \rho) \models_{\text{ML}} \Delta_S(\pi)$, which completes the proof.

■

Another property useful in the sequel is stated below.

Lemma 4.2.3 *$\mathcal{S} \models_{\text{RL}} \varphi \Rightarrow \varphi'$ iff $\mathcal{S}^\Delta \models_{\text{RL}} \varphi \Rightarrow \varphi'$.*

Proof ($\Rightarrow$) From $\mathcal{S} \models_{\text{RL}} \varphi \Rightarrow \varphi'$ (cf. Defintion 2.4.5) we obtain that for each terminating configuration $\gamma \in \mathcal{T}_{Cfg}$ and valuation $\rho : \mathit{Var} \rightarrow \mathcal{T}$ such that $(\gamma, \rho) \models_{\text{ML}} \varphi$, there is $\gamma' \in \mathcal{T}_{Cfg}$ and a path from γ to γ', denoted $\gamma \overset{\alpha^*}{\Longrightarrow}_{\mathcal{S}} \gamma'$, with $\alpha^* \in \mathcal{S}^*$, such that $(\gamma', \rho) \models \varphi'$. We prove by induction on the length of the sequence α^* that $\gamma \overset{\alpha^*}{\Longrightarrow}_{\mathcal{S}^\Delta} \gamma'$, which implies the ($\Rightarrow$) direction.

The base case (i.e. the length of the path is 0) is trivial. For the inductive step, we use the fact that each transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma_2$ is induced by semantical rule $\alpha \triangleq \pi \wedge \phi \Rightarrow \varphi \in \mathcal{S}$ and a valuation $\rho : \mathit{Var} \rightarrow \mathcal{T}$. Then, the corresponding rule $\pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \in \mathcal{S}^\Delta$ induces a corresponding transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}^\Delta} \gamma_2$ with the same valuation ρ.

($\Leftarrow$) In this case the proof is almost similar with the previous one, the only difference being in the inductive step where each transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}^\Delta} \gamma_2$ is induced by a semantical rule $\alpha^\Delta \triangleq \pi \Rightarrow \Delta_{\mathcal{S}}(\pi) \in \mathcal{S}^\Delta$ with a valuation $\rho : \mathit{Var} \rightarrow \mathcal{T}$. Then, the rule $\alpha \triangleq \pi \wedge \phi \Rightarrow \varphi \in \mathcal{S}$ induces a corresponding transition $\gamma_1 \overset{\alpha}{\Longrightarrow}_{\mathcal{S}} \gamma_2$ with the same valuation ρ.

■

Lemma 4.2.4 *A pattern φ is derivable for $\mathcal{S}$ iff φ is derivable for $\mathcal{S}^\Delta$.*

Proof By Definition 4.2.2, $\varphi \triangleq \pi \wedge \phi$ derivable for $\mathcal{S}$ means that $\Delta_{\mathcal{S}}(\varphi)$ is not the empty disjunction. This holds if and only if there exists a

$$\text{[SymbolicStep]}\ \frac{\varphi \textit{ derivable for } \mathcal{S}}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \Delta_{\mathcal{S}}(\varphi)}$$

$$\text{[CircHypothesis]}\ \frac{\alpha \in G \quad \alpha \textit{ covers } \varphi}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \Delta_{\{\alpha\}}(\varphi)}$$

$$\text{[Implication]}\ \frac{\models_{\text{ML}} \varphi \rightarrow \varphi'}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'}$$

$$\text{[CaseAnalysis]}\ \frac{\mathcal{S} \cup G \Vdash \varphi_1 \Rightarrow \varphi \quad \mathcal{S} \cup G \Vdash \varphi_2 \Rightarrow \varphi}{\mathcal{S} \cup G \Vdash (\varphi_1 \vee \varphi_2) \Rightarrow \varphi}$$

$$\text{[Transitivity]}\ \frac{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'' \quad \mathcal{S} \cup G \Vdash \varphi'' \Rightarrow \varphi'}{\mathcal{S} \cup G \Vdash \varphi \Rightarrow \varphi'}$$

Figure 4.6: Proof System for $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$.

rule $\pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in \mathcal{S}$ such that π_1 matches π. The proof is finished by noting that π_1 is also the left-hand side of the rule corresponding $\pi_1 \Rightarrow \Delta_{\mathcal{S}}(\pi_1) \in \mathcal{S}^{\Delta}$.
■

We now have almost all the ingredients for proving RL formulas by symbolic execution. In the following we assume a language with a semantics $\mathcal{S}$, and a finite of RL formulas with elementary patterns in their left-hand sides $G = \{\varphi_i \Rightarrow \varphi_i' \mid i = 1, \ldots, n\}$.

We say that a RL formula $\varphi \Rightarrow \varphi'$ is *derivable* for $\mathcal{S}$ if φ is derivable for $\mathcal{S}$. If G is a set of RL formulas then $\Delta_{\mathcal{S}}(G)$ is the set $\{\Delta_{\mathcal{S}}(\varphi) \Rightarrow \varphi' \mid \varphi \Rightarrow \varphi' \in G\}$, $\mathcal{S} \Vdash G$ denotes the conjunction $\bigwedge_{\varphi \Rightarrow \varphi' \in G} \mathcal{S} \Vdash \varphi \Rightarrow \varphi'$, and $\mathcal{S} \models_{\text{RL}} G$ denotes $\bigwedge_{\varphi \Rightarrow \varphi' \in G} \mathcal{S} \models_{\text{RL}} \varphi \Rightarrow \varphi'$. The proof system $\Vdash$ is shown in Figure 4.6. The following theorem establishes the soundness of our approach. It is based on a *circularity principle* also encountered in other coinductive frameworks, e.g., [95].

Theorem 4.2.1 (Circularity Principle) *If $\mathcal{S}$ is total and weakly well-defined, and G is derivable for $\mathcal{S}$, then $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ implies $\mathcal{S} \models_{\text{RL}} G$.*

Proof For all $i = 1, \ldots, n$ we apply the [Transitivity] rule of the original

RL proof system (Figure 2.1), with $\varphi_i'' \triangleq \Delta_{\mathcal{S}}(\varphi_i)$, and we obtain:

$$\frac{\mathcal{S} \vdash_G \varphi_i \Rightarrow \Delta_{\mathcal{S}}(\varphi_i) \quad (\mathcal{S} \cup G) \vdash \Delta_{\mathcal{S}}(\varphi_i) \Rightarrow \varphi_i'}{\mathcal{S} \vdash_G \varphi_i \Rightarrow \varphi_i'}$$

The first hypothesis: $S \vdash_G \varphi_i \Rightarrow \Delta_{\mathcal{S}}(\varphi_i)$ holds thanks to Lemma 4.2.1. The second one, $(\mathcal{S} \cup G) \vdash \Delta_{\mathcal{S}}(\varphi_i) \Rightarrow \varphi_i'$ holds because all the rules of $\Vdash$ are derived rules of $\vdash$, thanks to Lemma 4.2.1 again. Hence, we obtain $S \vdash_G \varphi_i \Rightarrow \varphi_i'$ for all $i = 1, \ldots, n$, i.e., $S \vdash_G G$. Then we obtain $S \vdash G$ by applying the derived rule *Set Circularity* of RL. Finally, the soundness of $\vdash$ (with the hypothesis that $\mathcal{S}$ is weakly well defined) implies $\mathcal{S} \models_{\text{RL}} G$. ■

If a semantics $\mathcal{S}$ is only total, but not weakly well-defined, one can use Theorem 4.2.1 to prove $\mathcal{S}^{\Delta} \models_{\text{RL}} G$ instead of $\mathcal{S} \models_{\text{RL}} G$. Indeed, $\mathcal{S}^{\Delta}$ is both total and weakly well-defined (thanks to Lemma 4.2.2) thus it satisfies the hypotheses of Theorem 4.2.1. Lemma 4.2.3 then ensures that by establishing $\mathcal{S}^{\Delta} \models_{\text{RL}} G$ we also proved $\mathcal{S} \models_{\text{RL}} G$, which is what we wanted in the first place.

Example 4.2.1 *In this example we show how the RL formula (4.3) is proved using the proof system shown in Figure 4.6, which amounts to verifying that the* `gcd` *program meets its specification. Instead of (4.3) we write* $\varphi^{\texttt{gcd}} \Rightarrow \varphi'^{\texttt{gcd}}$*, where:*

$$\varphi^{\texttt{gcd}} \triangleq \langle \texttt{gcd} \rangle_{\texttt{k}} \langle \texttt{a} \mapsto l_{\texttt{a}}\, \texttt{b} \mapsto l_{\texttt{b}}\, \texttt{x} \mapsto l_{\texttt{x}}\, \texttt{y} \mapsto l_{\texttt{y}}\, \texttt{r} \mapsto l_{\texttt{r}} \rangle_{\texttt{env}} \\ \langle l_{\texttt{a}} \mapsto a\, l_{\texttt{b}} \mapsto b\, l_{\texttt{x}} \mapsto x\, l_{\texttt{y}} \mapsto y\, l_{\texttt{r}} \mapsto r \rangle_{\texttt{store}}\ \sigma \wedge a \geq 0 \wedge b \geq 0,$$

and

$$\varphi'^{\texttt{gcd}} \triangleq (\exists x')(\exists y')(\exists r') \langle \cdot \rangle_{\texttt{k}} \langle \texttt{a} \mapsto l_{\texttt{a}}\, \texttt{b} \mapsto l_{\texttt{b}}\, \texttt{x} \mapsto l_{\texttt{x}}\, \texttt{y} \mapsto l_{\texttt{y}}\, \texttt{r} \mapsto l_{\texttt{r}} \rangle_{\texttt{env}} \\ \langle l_{\texttt{a}} \mapsto a\, l_{\texttt{b}} \mapsto b\, l_{\texttt{x}} \mapsto x'\, l_{\texttt{y}} \mapsto y'\, l_{\texttt{r}} \mapsto r' \rangle_{\texttt{store}}\ \sigma \wedge x' = gcd(a, b).$$

Note that we use σ *to denote the rest of the configuration, where each cell contain a variable. The pattern* $\varphi^{\texttt{gcd}}$ *contains a loop, and in order for the proof to succeed, we need an additional RL formula as loop invariant. Thus, we consider the RL formula* $\varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}$*, where:*

$$\varphi^{\texttt{while}} \triangleq \langle \texttt{while}\,(\texttt{y} > 0)\,\{\texttt{r} = \texttt{x}\%\texttt{y};\ \texttt{x} = \texttt{y};\ \texttt{y} = \texttt{r};\ \} \rangle_{\texttt{k}} \\ \langle \texttt{a} \mapsto l_{\texttt{a}}\, \texttt{b} \mapsto l_{\texttt{b}}\, \texttt{x} \mapsto l_{\texttt{x}}\, \texttt{y} \mapsto l_{\texttt{y}}\, \texttt{r} \mapsto l_{\texttt{r}} \rangle_{\texttt{env}} \\ \langle l_{\texttt{a}} \mapsto a\, l_{\texttt{b}} \mapsto b\, l_{\texttt{x}} \mapsto x\, l_{\texttt{y}} \mapsto y\, l_{\texttt{r}} \mapsto r \rangle_{\texttt{store}}\ \sigma \\ \wedge\, gcd(a, b) =_{Int} gcd(x, y) \wedge x \geq 0 \wedge y \geq 0,$$

and

$$\begin{aligned}\varphi'^{\mathtt{while}} \triangleq\ & (\exists x')(\exists y')(\exists r')\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x'\, l_{\mathtt{y}} \mapsto y'\, l_{\mathtt{r}} \mapsto r'\rangle_{\mathsf{store}}\ \sigma \\ & \wedge\ gcd(a,b) =_{Int} gcd(x',y') \wedge x' \geq 0 \wedge y' \geq 0.\end{aligned}$$

The rule $\varphi^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}}$ *says that the* `while` *loop preserves the invariant: the gcd of the values corresponding to program variables* `a`, `b` *equals the gcd of the values of variables* `x`, `y`. *At this point, the set* G *includes the rules* $\varphi^{\mathtt{gcd}} \Rightarrow \varphi'^{\mathtt{gcd}}$ *and* $\varphi^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}}$.

Now we can prove $\varphi^{\mathtt{gcd}} \Rightarrow \varphi'^{\mathtt{gcd}}$ *by applying the deductive system* $\Vdash$ *to the set of goals* G *as follows:*

- *We apply* [SymbolicStep] *a finite number of times until we obtain* $\mathcal{S} \cup G \Vdash \varphi^{\mathtt{gcd}} \Rightarrow \varphi_1^{\mathtt{gcd}}$, *where*

  $$\begin{aligned}\varphi_1^{\mathtt{gcd}} \triangleq\ & \langle \mathtt{while}\,(\mathtt{y} > 0)\,\{\mathtt{r} = \mathtt{x}\%\mathtt{y};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{r};\ \}\rangle_{\mathsf{k}} \\ & \langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto a\, l_{\mathtt{y}} \mapsto b\, l_{\mathtt{r}} \mapsto r\rangle_{\mathsf{store}}\ \sigma \\ & \wedge\ gcd(a,b) =_{Int} gcd(x,y)\end{aligned}$$

- *Now we can apply* [CircHypoyhesis] *with* $\varphi^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}} \in G$, *and we obtain* $\mathcal{S} \cup G \Vdash \varphi_1^{\mathtt{gcd}} \Rightarrow \varphi_2^{\mathtt{gcd}}$, *where:*

  $$\begin{aligned}\varphi_2^{\mathtt{gcd}} \triangleq\ & (\exists x')(\exists y')(\exists r')\langle\cdot\rangle_{\mathsf{k}} \\ & \langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x'\, l_{\mathtt{y}} \mapsto y'\, l_{\mathtt{r}} \mapsto r'\rangle_{\mathsf{store}}\ \sigma \\ & \wedge\ gcd(a,b) =_{Int} gcd(x',y') \wedge x' \geq 0 \wedge y' \geq 0\end{aligned}$$

- *Next, since* $\varphi_2^{\mathtt{gcd}}$ *and* $\varphi'^{\mathtt{while}}$ *are essentially the same pattern, then* $\models_{\mathrm{ML}} \varphi_2^{\mathtt{gcd}} \rightarrow \varphi'^{\mathtt{while}}$. *So, we apply* [Implication] *and we obtain* $\mathcal{S} \cup G \Vdash \varphi_2^{\mathtt{gcd}} \Rightarrow \varphi'^{\mathtt{gcd}}$.

A graphical representation of the proof tree for the formula (4.3) is shown below ($\varphi \overset{\mathsf{Rule}}{\Longrightarrow} \varphi'$ *represents* $\dfrac{\cdots}{\varphi \Rightarrow \varphi'}$ [Rule]*):*

$$\varphi^{\mathtt{gcd}} \xrightarrow{\mathit{SymbolicStep}^{+}} \varphi_1^{\mathtt{gcd}} \xrightarrow{\quad\mathit{CircHypothesis}\quad} \varphi_2^{\mathtt{gcd}} \xrightarrow{\mathit{Implication}} \varphi'^{\mathtt{gcd}}$$

Finally, we apply [Transitivity] *twice: first, we obtain* $\mathcal{S} \cup G \Vdash \varphi^{\mathtt{gcd}} \Rightarrow \varphi_2^{\mathtt{gcd}}$ *from* $\mathcal{S} \cup G \Vdash \varphi^{\mathtt{gcd}} \Rightarrow \varphi_1^{\mathtt{gcd}}$ *and* $\mathcal{S} \cup G \Vdash \varphi_1^{\mathtt{gcd}} \Rightarrow \varphi_2^{\mathtt{gcd}}$*, and then we obtain* $\mathcal{S} \cup G \Vdash \varphi^{\mathtt{gcd}} \Rightarrow \varphi'^{\mathtt{gcd}}$ *from* $\mathcal{S} \cup G \Vdash \varphi^{\mathtt{gcd}} \Rightarrow \varphi_2^{\mathtt{gcd}}$ *and* $\mathcal{S} \cup G \Vdash \varphi_2^{\mathtt{gcd}} \Rightarrow \varphi'^{\mathtt{gcd}}$.

Below we show the proof of the invariant rule $\varphi^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}}$*:*

- *By applying* [SymbolicStep] *several times and* [CaseAnalysis] *we obtain either* $S \cup G \Vdash \varphi^{\mathtt{while}} \Rightarrow \varphi_1^{\mathtt{while}}$*, where*

$$\begin{aligned}\varphi_1^{\mathtt{while}} \triangleq\ & \langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x\, l_{\mathtt{y}} \mapsto y\, l_{\mathtt{r}} \mapsto r\rangle_{\mathsf{store}}\ \sigma \\ & \wedge gcd(a,b) =_{Int} gcd(x,y) \wedge x \geq_{Int} 0 \wedge y =_{Int} 0\end{aligned}$$

  or $S \cup G \Vdash \varphi^{\mathtt{while}} \Rightarrow \varphi_2^{\mathtt{while}}$*, where*

$$\begin{aligned}\varphi_2^{\mathtt{while}} \triangleq\ & \langle \{\mathtt{r} = \mathtt{x}\%\mathtt{y};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{r};\ \} \curvearrowright \mathtt{while}\,(\mathtt{y} > 0)\,\{\ldots\}\rangle_{\mathsf{k}} \\ & \langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x\, l_{\mathtt{y}} \mapsto y\, l_{\mathtt{r}} \mapsto r\rangle_{\mathsf{store}}\ \sigma \\ & \wedge gcd(a,b) =_{Int} gcd(x,y) \wedge x \geq_{Int} 0 \wedge y >_{Int} 0.\end{aligned}$$

- *We observe that* $\varphi_1^{\mathtt{while}}$ *implies* $\varphi'^{\mathtt{while}}$ *(because there are* $x' = x$*,* $y' = 0$*, and* $r' = r$ *such that* $\varphi'^{\mathtt{while}}$ *holds). Thus, we first apply* [Implication] *to obtain* $S \cup G \Vdash \varphi_1^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}}$*, and then, since* $S \cup G \Vdash \varphi^{\mathtt{while}} \Rightarrow \varphi_1^{\mathtt{while}}$*, we have* $S \cup G \Vdash \varphi^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}}$ *by* [Transitivity].

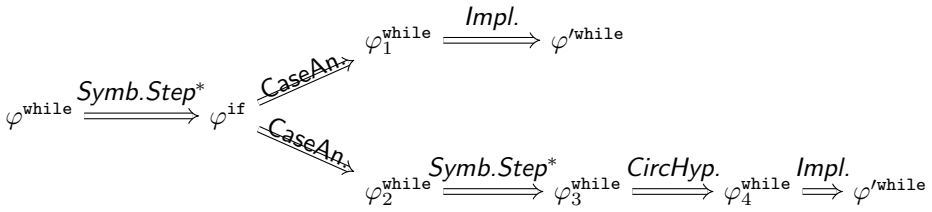
- *On the other hand, we can apply* [SymbolicStep] *over* $\varphi_2^{\mathtt{while}}$ *until we obtain* $S \cup G \Vdash \varphi_2^{\mathtt{while}} \Rightarrow \varphi_3^{\mathtt{while}}$*, where*

$$\begin{aligned}\varphi_3^{\mathtt{while}} \triangleq\ & \langle \mathtt{while}\,(\mathtt{y} > 0)\,\{\mathtt{r} = \mathtt{x}\%\mathtt{y};\ \mathtt{x} = \mathtt{y};\ \mathtt{y} = \mathtt{r};\ \}\rangle_{\mathsf{k}} \\ & \langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto y\, l_{\mathtt{y}} \mapsto r\, l_{\mathtt{r}} \mapsto x\%y\rangle_{\mathsf{store}}\ \sigma \\ & \wedge gcd(a,b) =_{Int} gcd(x,y) \wedge x \geq_{Int} 0 \wedge y >_{Int} 0\end{aligned}$$

- *Now, since* $\varphi^{\mathtt{while}} \Rightarrow \varphi'^{\mathtt{while}}$ *covers* $\varphi_3^{\mathtt{while}}$ *we apply* [CircHypothesis] *and we get* $S \cup G \Vdash \varphi_3^{\mathtt{while}} \Rightarrow \varphi_4^{\mathtt{while}}$*, where the pattern* $\varphi_4^{\mathtt{while}}$ *is*

$$\begin{aligned}\varphi_4^{\mathtt{while}} \triangleq\ & (\exists x')(\exists y')(\exists r')\langle\cdot\rangle_{\mathsf{k}}\langle \mathtt{a} \mapsto l_{\mathtt{a}}\, \mathtt{b} \mapsto l_{\mathtt{b}}\, \mathtt{x} \mapsto l_{\mathtt{x}}\, \mathtt{y} \mapsto l_{\mathtt{y}}\, \mathtt{r} \mapsto l_{\mathtt{r}}\rangle_{\mathsf{env}} \\ & \langle l_{\mathtt{a}} \mapsto a\, l_{\mathtt{b}} \mapsto b\, l_{\mathtt{x}} \mapsto x'\, l_{\mathtt{y}} \mapsto y'\, l_{\mathtt{r}} \mapsto r'\rangle_{\mathsf{store}}\ \sigma \\ & \wedge gcd(a,b) =_{Int} gcd(x',y') \wedge x' \geq_{Int} 0 \wedge y' >_{Int} 0\end{aligned}$$

The proof tree for the formula $\varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}$ *is*



As in the previous cases, we apply [Implication] *to deduce* $S \cup G \Vdash \varphi_4^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}$*, and then by a few applications of* [Transitivity] *we obtain* $\varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}$.

This concludes the proof of the set of goals $\{\varphi^{\texttt{gcd}} \Rightarrow \varphi'^{\texttt{gcd}}, \varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}\}$ *, and, in particular, of the fact that* `gcd` *meets its specification (4.3). Note how the proofs of all goals have used symbolic execution as well other goals as circular hypotheses. Moreover, the proof obligation stating that the loop's body satisfies the invariant, which would be required in Hoare logic, is no longer necessary since it is implicitly proved by the symbolic steps and a circular application of the rule specifying the loop.*

4.2.1 Default strategy for automation

Our proof system still leaves some freedom regarding the order of rule applications and still requires some creative user input when, e.g, choosing the patterns $\varphi_1, \varphi_2, \varphi''$ in its rules. We define the following strategy (already applied in the previous example) in order to completely automate proof searches. In the default strategy, the rules are applied with the following priorities (note that all the rules are applied "bottom-up"):

1. Implication has highest priority. It is only applied for closing proof branches;

2. CircularHypothesis, followed by all the applications of CaseAnalysis required to break disjunctive patterns into elementary patterns, has second priority;

3. SymbolicStep, also followed by as many applications of CaseAnalysis as needed to break disjunctive patterns into elementary ones, has third priority.

Note that Transitivity is not restricted in any way. In fact, it is just used implicitly for building larger proofs from smaller proofs steps. We call the above strategy the *default strategy*. It transforms proof attempts of $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ into the building of the symbolic transition relation $\Rightarrow^{\mathfrak{s}}_{\mathcal{S} \cup G}$, which is done by our symbolic execution tool [5]. More details on the implementation are given in the Section 5.1.

4.2.2 Weak Completeness: disproving RL formulas

It is a good idea to try to disprove RL formulas as well as to try to prove them. The default strategy of our proof system can also be used for disproving formulas: if it terminates in by failing to prove its input (a set of RL formulas, with some reasonable restrictions presented below) then the input in question is invalid.

A rule $\varphi \Rightarrow \varphi'$ is *terminal* if φ' is non-derivable for $\mathcal{S} \cup G$. For example, the RL formulas $\varphi^{\texttt{gcd}} \Rightarrow \varphi'^{\texttt{gcd}}$ and $\varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}$ (Example 4.2.1) are terminal, because their right-hand sides contain empty code that cannot be executed further. The specification of a program, like $\varphi^{\texttt{gcd}} \Rightarrow \varphi'^{\texttt{gcd}}$ for `gcd`, is typically terminal because it refers to what the program computes when it terminates. Auxiliary formulas, like $\varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}$, may or may not be terminal.

A RL formula $\varphi \Rightarrow \varphi'$ is *terminating* if all configurations $\gamma \in [\![\varphi]\!]$ are terminating, and a set of formulas is terminating iff every formula in it is terminating. For example, $\{\varphi^{\texttt{gcd}} \Rightarrow \varphi'^{\texttt{gcd}}, \varphi^{\texttt{while}} \Rightarrow \varphi'^{\texttt{while}}\}$ is terminating. A set $\mathcal{S}$ of RL formulas is *confluent* if the transition relation $\Rightarrow^*_{\mathcal{S}}$ is confluent (i.e. the terms can be rewritten in more than one way, but the result is always the same).

The next theorem is our weak completeness result. It assumes a situation where, on a given proof branch for a terminal goal ($\varphi \Rightarrow \varphi'$), the default strategy of our proof system is "stuck"; thus, its execution terminates in failure.

Theorem 4.2.2 (Weak Completeness) *Consider a confluent set of RL formulas $\mathcal{S}$, a set of terminating formulas $G = \{\pi_i \wedge \phi_i \Rightarrow \pi'_i \wedge \phi'_i | i \in I\}$, a terminal formula $\varphi \Rightarrow \varphi' \in G$, and a proof branch of $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(G)$ generated by the default strategy, starting from $\mathcal{S} \cup G \Vdash \Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi')$ and ending in $\mathcal{S} \cup G \Vdash \varphi'' \Rightarrow \varphi'$, such that φ'' is not derivable for $\mathcal{S} \cup G$ and*

$\not\models_{\mathrm{ML}} \varphi'' \to \varphi'$. *Then* $\mathcal{S} \not\models_{\mathrm{RL}} G$.

Proof From the proof branch we extract a symbolic execution $[\varphi]_\sim \Rightarrow^{\mathfrak{S}*}_{\mathcal{S}\cup G} \varphi''_\sim$. Let $\gamma'' \in [\![\varphi'']\!] \setminus [\![\varphi']\!]$. A (feasible) symbolic execution is simulated by a concrete execution $\gamma \Rightarrow^*_{\mathcal{S}\cup G} \gamma''$ with $\gamma \in [\![\varphi]\!]$, thanks to Corollary 3.3.2. We assume (by contradiction) that $S \models_{\mathrm{RL}} G$.

First, we show ($\diamondsuit$): $\gamma \Rightarrow^*_{\mathcal{S}} \gamma''$. For this, we show that for every step, say, $\gamma_1 \Rightarrow_{\{\alpha\}} \gamma_2$ with $\alpha \in G$, there is an execution with rules in $\mathcal{S}$, i.e., $\gamma_1 \Rightarrow^*_{\mathcal{S}} \gamma_2$. Then we replace every such step $\gamma_1 \Rightarrow_{\{\alpha\}} \gamma_2$ in $\gamma \Rightarrow^*_{\mathcal{S}\cup G} \gamma''$ ($\alpha \in G$) by the corresponding execution $\gamma_1 \Rightarrow^*_{\mathcal{S}} \gamma_2$ and obtain the desired execution $\gamma \Rightarrow^*_{\mathcal{S}} \gamma''$.

We now prove ($\spadesuit$) $\gamma_1 \Rightarrow^*_{\mathcal{S}} \gamma_2$ from $\gamma_1 \Rightarrow_{\{\alpha\}} \gamma_2$ with $\alpha \in G$. From the assumption $\mathcal{S} \models_{\mathrm{RL}} G$ we obtain $\mathcal{S} \models_{\mathrm{RL}} \alpha (= \pi_1 \wedge \phi_1 \Rightarrow \pi_2 \wedge \phi_2 \in G)$. From $\gamma_1 \Rightarrow_{\{\alpha\}} \gamma_2$ we get $(\gamma_1, \rho) \models_{\mathrm{ML}} \pi_1 \wedge \phi_1$ for some valuation ρ and $(\gamma_2, \rho) \models_{\mathrm{ML}} \pi_2 \wedge \phi_2$. Thus, $\gamma_2 = \rho(\pi_2)$. From $\mathcal{S} \models_{\mathrm{ML}} \alpha$ and $(\gamma_1, \rho) \models_{\mathrm{ML}} \pi_1 \wedge \phi_1$ (note that γ_1 is terminating, since G is terminating) we get that there exists γ'_2 such that $\gamma_1 \Rightarrow^*_{\mathcal{S}} \gamma'_2$ and $(\gamma'_2, \rho) \models_{\mathrm{ML}} \pi_2 \wedge \phi_2$. In particular, $\gamma'_2 = \rho(\pi_2)$. Thus, $\gamma'_2 = \gamma_2$, so $\gamma_1 \Rightarrow^*_S \gamma_2$. ($\spadesuit$) is proved, and so is ($\diamondsuit$).

We thus have $\gamma \Rightarrow^*_{\mathcal{S}} \gamma''$, where $\gamma'' \in [\![\varphi'']\!] \setminus [\![\varphi']\!]$, and γ terminating (since $\gamma \in [\![\varphi]\!]$ and G is terminating). We have assumed $\mathcal{S} \models_{\mathrm{RL}} G$, in particular, $\mathcal{S} \models_{\mathrm{RL}} \varphi \Rightarrow \varphi'$. This means that from the (terminating) γ, there is $\gamma''' \in [\![\varphi']\!]$ and $\gamma \Rightarrow^*_{\mathcal{S}} \gamma'''$. On the other hand, γ'' is terminal (because φ'' is non-derivable) and $\gamma'' \in [\![\varphi'']\!]$, so we obtain thanks to the confluence hypothesis that any execution starting in γ ends up in γ''. Thus, the successor $\gamma''' \in [\![\varphi']\!]$ of γ is on such an execution. There are two cases:

- $\gamma''' = \gamma''$: impossible because $\gamma'' \in [\![\varphi'']\!] \setminus [\![\varphi']\!]$ (hypothesis).
- γ''' strictly precedes γ''. This is also impossible (since γ''' has successors), but φ' is non-derivable (since the goal $\varphi \Rightarrow \varphi'$ we started with is terminal).

The contradiction was generated by our assumption $\mathcal{S} \models_{\mathrm{RL}} G$, so, we conclude $\mathcal{S} \not\models_{\mathrm{RL}} G$.

■

Note that the default strategy is "stuck" after the last sequent $\mathcal{S} \cup G \Vdash \varphi'' \Rightarrow \varphi'$ because no rule can be applied from there on, in the current proof

branch: Implication cannot be applied to close the proof branch because $\not\models_{\text{ML}} \varphi'' \rightarrow \varphi'$, CircularHypothesis and SymbolicStep cannot be applied since φ'' is non-derivable, and CaseAnalysis cannot be applied because φ'' is an elementary pattern.

Remark 4.2.4 *Theorem 4.2.2 is proved by showing that there are configurations γ, γ'' such that $\gamma \Rightarrow_{\mathcal{S}}^{*} \gamma''$, $\gamma \in [\![\varphi]\!]$, and $\gamma'' \in [\![\varphi'']\!] \setminus [\![\varphi']\!]$. Using confluence, all executions starting in γ end up in the (terminal) γ'', and using the fact that $\varphi \Rightarrow \varphi'$ is terminal, no configuration on any of these executions may encounter $[\![\varphi']\!]$. If the confluence of $\mathcal{S}$ and/or the fact that G is terminating do not hold, one can attempt, as an alternative approach for establishing weak completeness, to use model checking starting from γ, in order to check whether or not there is a reachable configuration in $[\![\varphi']\!]$. The terminating nature of G ensures the model checking will always terminate.*

Together with the soundness result, weak completness says that when our proof system's default strategy terminates (either successfully, by proving all the goals, our unsuccessfully, by getting stuck on a given proof branch), it correctly solves the problem of whether the goals given to it as input are valid or not. That is, if it terminates, our approach correctly solves the program-verification problem.

Chapter 5

Symbolic execution within $\mathbb{K}$ Framework

In this chapter we present a prototype tool implementing our symbolic execution approach. We first briefly describe the tool and its integration within the $\mathbb{K}$ framework (Section 5.1). Then we illustrate the most significant features of the tool by the means of use cases involving nontrivial languages and programs (Section 5.2). In the rest of the chapter we present a verification tool developed on top of the symbolic execution framework, which is used to perform program verification using Reachability Logic (Section 5.3).

5.1 Symbolic Execution within the $\mathbb{K}$ Framework

Our tool is part of $\mathbb{K}$ [97, 109], a rewrite-based executable semantics framework in which programming languages, type systems, and formal analysis tools can be defined. Beside some toy languages used for teaching, there are a few real-life programming languages, supporting different paradigms, that have been successfully defined in $\mathbb{K}$ (e.g. C [35], Java [17] - see Section 2.5 for details).

In $\mathbb{K}$, the definition of a programming language, say, $\mathcal{L}$, is compiled into a rewrite theory. Then, the $\mathbb{K}$ runner executes programs in $\mathcal{L}$ by applying the resulting rewrite rules to configurations containing programs. Our implementation follows the same process. The main difference is that our

new $\mathbb{K}$ compiler includes some transformations steps: rule linearisation, replacing the data subterms in a rule with variables and add them into the rule's condition, add a configuration cell for path condition, and modify rules such that the path condition is updated appropriately at runtime. The effect is that the compiled rewrite theory we obtain defines the so-called symbolic semantics of $\mathcal{L}$ instead of its concrete semantics. We note that the symbolic semantics can execute programs with concrete inputs as well, and thus, the initial semantics of $\mathcal{L}$ is not affected. For user convenience we have also improved the $\mathbb{K}$ runner with some specific options which are useful for providing programs with symbolic input and setting up the initial path condition.

By choosing the $\mathbb{K}$ framework as a basis for our tool we take the full advantage of its language-independent capabilities. Given the $\mathbb{K}$ definition of a programming language we can generate fully automatically the symbolic semantics, and then run programs with symbolic values. Therefore, users which already have a $\mathbb{K}$ definition of a language can perform symbolic execution with no additional effort.

The main part of the tool was written in the same language as the $\mathbb{K}$ tool, namely Java. There are also several pieces of code for interaction with the Z3 SMT solver written in $\mathbb{K}$ itself. In the following, we are going to describe in detail the options of the tool. For convenience, we assume that a file named `cink.k` contains the $\mathbb{K}$ definition of CinK (Section 2.5) and CinK programs are stored in files having the `*.cink` extension (e.g. `sum.cink`).

5.1.1 Compiling the symbolic semantics

In order to compile the symbolic semantics of CinK one must type the following command:

```
$ kompile cink.k --backend symbolic
```

This option enables the following transformations:

- *Generate symbolic values support.* This step is responsible for generating the set *Var* of symbolic values and also the predicates which state their membership to different sorts.

- *Add input support for symbolic values.* This transformation searches the configuration for cells which are connected to the input stream and generates an input variable `$IN` which can be initialised with symbolic values when running programs.

- *Add cell for path condition.* To keep track of the current path condition we use the $\mathbb{K}$ configuration of the language which is enriched with a new cell called `path-condition`. This gives us the freedom to handle the path condition in a language-independent manner and also brings us the advantage of attaching separate path conditions to different result configurations.

- *Rule linearisation.* In practice, the $\mathbb{K}$ rewrite rules are not linear (i.e. they contain more than once the same variable in the left hand side) as we assume in Chapter 3. This step turns them into linear rules by replacing duplicate variables with fresh ones and adding corresponding equalities in the rule's condition.

- *Add rule condition to path condition.* This transformation consists in modifying the rule such that the path condition cell is updated with the rule's condition. If $l \wedge b \Rightarrow r$ is our current rule and ψ the current path condition, then the new path condition will be $\psi \wedge b$ if $\psi \wedge b$ is not unsatisfiable.

The last transformation step is probably the most important. It involves a call to the SMT solver for checking the satisfiability of the future state of the program. If the solver finds it unsatisfiable then the rule is not applied because it makes no sense to investigate an infeasible execution path. However, the rule will apply when the solver is not able to figure out if the formula is satisfiable or not (i.e. returns *unknown*), since otherwise the tool can miss feasible execution paths. Another discussion related to the SMT call is how the tool deals with the rule condition b when it contains expressions which cannot be sent to the solver, like inductive definitions and predicates used by the $\mathbb{K}$ tool for matching. For that, we provide an additional step which splits b into $b_1 \wedge b_2$, where b_1 is the part which is sent to the solver, and b_2 is the part related to matching. The condition b is transformed into its conjunctive normal form and then the split is done using a tagging mechanism which identifies which clauses

are actual matching predicates. After the split, the new path condition becomes $\psi \wedge b_1$ instead of $\psi \wedge b$ and the new rule condition is b_2, which is relevant for matching purposes. On the other hand, the user must provide by hand the inductive definitions (if any) in $\mathbb{K}$. Let us consider that *gcd*, given using an inductive definition in $\mathbb{K}$ is part of b_1. The tool is able to simplify b_1 before sending it to the solver, by applying the definition of *gcd*. If the formula reduces to false, then it is unsatisfiable and for sure the new path is unfeasible and will not be explored. If not, b_1 is sent to the solver as it is, *gcd* being translated as an uninterpreted function. However, a question arises here: can we still trust the response given by the solver when *gcd* can be interpreted by the solver as *any* function? If the solver says that the formula is unsatisfiable then we know for sure that for any interpretation of *gcd* (including the real greatest common divisor function) it remains unsatisfiable. On the other hand, if the solver says that the formula is satisfiable then we do not know for sure that it is also satisfiable for the appropriate interpretation of *gcd*. In this case the tool will consider the formula satisfiable and will continue to explore this execution path. In some sense, because the solver is not able to deal with all types of formulas we are interested in, the tool over-approximates the space of feasible paths.

5.1.2 Running programs with symbolic input

The CinK language also allows I/O interactions. This enables the execution of CinK programs with symbolic input data. We compile the resulting definition by calling the $\mathbb{K}$ compiler with `--backend symbolic` option as discussed in Section 5.1.1. Now, programs such as `sum.cink` (shown in Figure 5.1) can be run with the $\mathbb{K}$ runner in the following ways:

1. with concrete inputs;
2. with symbolic inputs;
3. on one arbitrary execution path, or on all paths up to a given bound;
4. in a step-wise manner, or by letting the program completely execute a given number of paths.

```
void main() {
  int n, s;
  n = read();
  s = 0;
  while (n > 0) {
    s = s + n;
    n = n - 1;
  }
  cout << "Sum = " << s;
}
```

Figure 5.1: `sum.cink`

For example, by running `sum.cink` with a symbolic input n (here and thereafter we use mathematical font for symbolic values) and requiring at most five completed executions, the $\mathbb{K}$ runner outputs the five resulting, final configurations, one of which is shown below, in a syntax slightly simplified for readability:

```
$ krun sum.cink -cIN=n -cPC="true"
  <k> . </k>
  <path-condition>
        n > 0 ∧ (n − 1 > 0) ∧ ¬((n − 1) − 1 > 0)
   </path-condition>
    <env>
             n |-> l_n
             s |-> l_s
    </env>
    <state>
             l_n |-> (n − 1) − 1
             l_s |-> n + (n − 1)
    </state>
  ...
```

The program is finished since the `k` cell has no code left to execute. The path condition actually means $n = 2$, and in this case the sum `s` equals $n + (n - 1) = 2 + 1$, as shown by the `state` cell. The `-cIN=`n option sends on the input stream of the program the symbolic value n, while `-cPC="true"` sets the initial path condition.

To get all possible solutions the tool has be run using `--search`:

```
$ krun sum.cink -cIN=n -cPC="true" --search
```

Since `sum.cink` contains a loop that depends on the symbolic value n, the command above will run forever. Still, users can bound the execution to a specific number of solutions:

```
$ krun sum.cink -cIN=n -cPC="true" --search --bound 3
  Solution 1
  ...
   <k> . </k>
  <path-condition>
        ¬(n > 0)
   </path-condition>
      ...

  Solution 2
  ...
   <k> . </k>
  <path-condition>
        n > 0 ∧ ¬(n − 1 > 0)
   </path-condition>
      ...

  Solution 3
  ...
   <k> . </k>
  <path-condition>
        n > 0 ∧ (n − 1 > 0) ∧ ¬((n − 1) − 1 > 0)
   </path-condition>
      ...
```

The command above shows three possible execution paths which compute the sums of numbers up to 0, 1, and 2 respectively. The `--bound 3` option bounds the number of solutions to three. The number of solutions

can be also bounded using the initial path condition:

```
$ krun sum.cink -cIN=n -cPC="0 ≤ n ∧ n < 3" --search
  Solution 1
  ...
   <k> . </k>
  <path-condition>
        ¬(n > 0)
   </path-condition>
     ...

  Solution 2
  ...
   <k> . </k>
  <path-condition>
        n > 0 ∧ ¬(n − 1 > 0)
   </path-condition>
     ...

  Solution 3
  ...
   <k> . </k>
  <path-condition>
        n > 0 ∧ (n − 1 > 0) ∧ ¬((n − 1) − 1 > 0)
   </path-condition>
     ...
```

Users can run the program in a step-wise manner using the $\mathbb{K}$ runner debugger in order to see intermediary configurations in addition to final ones. During this process they can interact with the runner, e.g., by choosing one execution branch of the program among several, feeding the program with inputs, or letting the program run on an arbitrarily chosen path until its completion.

5.2 Symbolic execution: use cases

In this section we illustrate some direct use cases for our symbolic execution tool. Moreover, we run our tool on multiple programming languages, already defined in $\mathbb{K}$, to emphasise that our tool is language independent. In Section 5.2.1 we show how the symbolic execution framework works together with the Maude LTL (Linear Temporal Logic) model checker [77]. Then, in Section 5.2.2, we show a simple example of bounded model checking over programs which manipulate symbolic arrays. Finally, we test our symbolic execution tool on an object oriented language and we use it to check the well-known virtual method call mechanism (Section 5.2.3).

5.2.1 LTL model checking

The $\mathbb{K}$ runner includes a hook to the Maude LTL (Linear Temporal Logic) model checker [77]. Thus, one can model check LTL formulas on programs having a finite state space (or by restricting the verification to a finite subset of the state space). This requires an (automatic) extension of the syntax and semantics of a language for including labels that are used as atomic propositions in the LTL formulas. Predicates on the program's variables can be used as propositions in the formulas as well, using the approach outlined in [71]. We have enriched the $\mathbb{K}$ definition of CinK with syntax and semantics for atomic propositions. Below is the LTL satisfaction relation for the atomic proposition `logInv(a,x,k)`:

$$B \models_{Ltl} \texttt{logInv}(A, X, K) \Rightarrow \mathit{true}\ \mathit{requires}\ \texttt{val}(B, X) *_{Int} \texttt{pow}(2, \texttt{val}(B, K)) \leq_{Int} \texttt{val}(B, A) \\ \wedge_{Bool}\ (\texttt{val}(B, A) <_{Int} (\texttt{val}(B, X) +_{Int} 1) *_{Int} \texttt{pow}(2, \texttt{val}(B, K)))$$

The variable B has sort *Cfg*, while variables A, B, and K are all identifiers, which are mapped in the configuration to integer values. The functions `pow` and `val` are additional functions used for computing the mathematical power function, and for retrieving the value of an identifier from the configuration, respectively. For each such rule, the tool automatically generates support for checking whether the current path condition implies the rule's condition. If this implication does not hold then the rule does not apply, since we are in a state where the LTL property does not hold.

```
void main() {
  int k, a, x;
  a = read();
  x = a;
  while (x > 1) {
    x = x / 2;
    k = k + 1;
    L : {}
  }
}
```

Figure 5.2: `log.cink`

Consider for instance the program `log.cink` in Figure 5.2, which computes the integer binary logarithm of an integer read from the input. We prove that whenever the loop visits the label `L`, the inequalities $\mathtt{x} * 2^{\mathtt{k}} \leq \mathtt{a} < (\mathtt{x}+1) * 2^{\mathtt{k}}$ hold. The invariant was guessed using several step-wise executions. We let `a` be a symbolic value and restrict it in the interval (0..10) to obtain a finite state space. We prove that the above property, denoted by `logInv(a,x,k)` holds whenever the label `L` is visited and `a` is in the given interval, using the following command (again, slightly edited for better readability):

```
$ krun log.cink -cPC="a >_Int 0  /\_Bool  a <_Int 10" -cIN="a"
                -ltlmc "□_Ltl (L →_Ltl logInv(a, x, k))"
true
```

The $\mathbb{K}$ runner executes the command by calling the Maude LTL model-checker for the LTL formula $\Box_{Ltl}\,(\mathtt{L} \rightarrow_{Ltl} \mathtt{logInv(a,x,k)})$ and the initial configuration having the program `log.cink` in the computation cell `k`, the symbolic value a in the input cell in, and the constraint $a >_{Int} 0 \wedge_{Bool} a <_{Int} 10$ in the path condition. The result returned by the tool is that the above LTL formula holds.

5.2.2 SIMPLE, symbolic arrays, and bounded model checking

In this section we illustrate symbolic arrays in the SIMPLE [102] language and we show how the $\mathbb{K}$ runner can directly be used for performing bounded model checking. SIMPLE is a non-trivial imperative programming language which includes multidimensional arrays and array refer-

```
void init(int[] a, int x, int j){
     int i = 0, n = sizeOf(a);
     a[j] = x;
     while (a[i] != x && i < n) {
       a[i] = 2 * i;
       i = i + 1;
     }
     if (i > j) {
       print("error");
     }
}
```

```
void main() {
     int n = read();
     int j = read();
     int x = read();
     int a[n], i = 0;
     while (i < n) {
       a[i] = read();
       i = i + 1;
     }
     init(a, x, j);
}
```

Figure 5.3: SIMPLE program: `init-arrays`

ences, functions and function values, blocks with local variables, exceptions, concurrency via dynamic thread creation/termination and synchronisation. Further details about SIMPLE and its $\mathbb{K}$ definition can be found in [102].

In the program in Figure 5.3, the `init` method assigns the value `x` to the array `a` at an index `j`, then fills the array with ascending even numbers until it encounters x in the array; it prints *error* if the index `i` went beyond `j` in that process. The array and the indexes `i`, `j` are parameters to the function, passed to it by the `main` function which reads them from the input. In [3] it has been shown, using model-checking and abstractions on arrays, that this program never prints *error*.

We obtain the same result by running the program with symbolic inputs and using the $\mathbb{K}$ runner as a bounded model checker:

```
$ krun init-arrays.simple -cPC="n >Int 0" -search -cIN="n j x a b c"
                          -pattern="<T> <out> error </out> B:Bag </T>"
Search results:
No search results
```

The initial path condition is $n >_{Int} 0$. The symbolic inputs for `n,j,x` are entered as n j x, and the array elements a b c are also symbolic. The `--pattern` option specifies a pattern to be searched in the final configuration: the text *error* should be in the configuration's output buffer. The above command thus performs a bounded model-checking with symbolic inputs (the bound is implicitly set by the number of array elements given as inputs - 3). The command above does not return any solution, meaning that the program will never print *error*.

The result was obtained using symbolic execution without any additional tools or techniques. We note that array sizes are symbolic as well, a feature that, to our best knowledge, is not present in other symbolic execution frameworks.

5.2.3 KOOL: testing virtual method calls on lists

Another example we consider for analysis is a program written in the KOOL [53] programming language. KOOL is a concurrent, dynamic, object oriented language inspired from Smalltalk [48], and besides the imperative features (assignments, conditional statements, loops with break and continue) it includes support for many familiar object oriented features (classes, methods, inheritance, dynamic dispatch, exceptions via try/-catch, ...).

In Figure 5.4 we show a KOOL program which implements lists and ordered lists of integers using arrays. We use symbolic execution to check the well-known virtual method call mechanism of object-oriented languages: the same method call, applied to two objects of different classes, may have different outcomes.

The `List` class implements (plain) lists. It has methods for creating, copying, and testing the equality of lists, as well as for inserting and deleting elements in a list. Figure 5.4 shows only insertion and deletion of elements. The class `OrderedList` inherits from `List`. It redefines the `insert` method in order to ensure that the sequences of elements in lists are sorted in increasing order. The `Main` class creates a list `l1`, initializes `l1` and an integer variable `x` with input values, copies `l1` to a list `l2` and then inserts and deletes `x` in `l1`. Finally it compares `l1` to `l2` element by element, and prints *error* if it finds them different.

We use symbolic execution to show that the above sequence of method calls results in different outcomes, depending on whether `l1` is a `List` or an `OrderedList`. We first try the case where `l1` is a `List`, by issuing the following command to the $\mathbb{K}$ runner:

```
$ krun  lists.kool   -search -cIN="e1 e2 x"
                     -pattern="<T> <out> error </out> B:Bag </T>"
 Solution 1, State 50:
<path-condition>
   e1 = x ∧Bool ¬Bool (e1 = e2)
```

```
class List {
  int a[10];
  int size, capacity;
  ...

  void insert (int x) {
    if (size < capacity) {
      a[size] = x;  ++size;
    }
  }

  void delete(int x) {
    int i = 0;
    while(i < size-1 && a[i] != x) {
      i = i + 1;
    }
    if (a[i] == x) {
      while (i < size - 1) {
        a[i] = a[i+1];
        i = i + 1;
      }
      size = size - 1;
    }
  }
  ...
}
```

```
class OrderedList extends List {
  ...
  void insert(int x){
    if (size < capacity) {
      int i = 0, k;
      while(i < size && a[i] <= x) {
        i = i + 1;
      }
      ++size; k = size - 1;
      while(k > i) {
        a[k] = a[k-1]; k = k - 1;
      }
      a[i] = x;
    }
  }
}
class Main {
  void Main() {
    List l1 = new List();
    ... // read elements of l1 and x
    List l2 = l1.copy();
    l1.insert(x); l1.delete(x);
    if (l2.eqTo(l1) == false) {
      print("error\n");
    }
  }
}
```

Figure 5.4: `lists.kool`: implementation of lists in KOOL

```
</path-condition>
...
```

The command initializes `l1` with two symbolic values (e_1, e_2) and sets the program variable `x` to the symbolic value x. It searches for configurations that contain *error* in the output. The tool finds one solution, with $e_1 = x$ and $e_1 \neq e_2$ in the path condition. Since `insert` of `List` appends x at the end of the list and deletes the first instance of x from it, `l1` consists of (e_2, x) when the two lists are compared, in contrast to `l2`, which consists of (e_1, e_2). The path condition implies that the lists are different.

The same command on the same program but where `l1` is an `OrderedList` finds no solution. This is because `insert` in `OrderedList` inserts an element in a unique place (up to the positions of the elements equal to it) in an ordered list, and `delete` removes either the inserted element or one with the same value. Hence, inserting and then deleting an element leaves an ordered list unchanged.

Thus, virtual method call mechanism worked correctly in the tested scenarios. An advantage of using our symbolic execution tool is that the condition on the inputs that differentiated the two scenarios was discovered by the tool. This feature can be exploited in other applications such as test-case generation.

5.3 A prototype for Reachability-Logic Verification

In this section we illustrate on a few examples a prototype tool which implements the default strategy (shown in Section 4.2.1) of the $\Vdash$ deductive system that we have defined (Figure 4.6), in order to verify $\mathcal{S} \models_{\text{RL}} G$, where $\mathcal{S}$ is a given language semantics and G a set of reachability formulas (goals). The implementation is part of the $\mathbb{K}$ tools suite [74] and it has been developed on top of our symbolic execution tool presented in Section 5.1.

In terms of implementation, our prototype reuses components of the $\mathbb{K}$ framework: parsing, compilation steps, support for symbolic execution, and connections to Maude's [77] state-space explorer and to the Z3 SMT solver [34]. Given a language definition $\mathcal{S}$ and a set of RL formulas G, `kcheck` build a new definition $\mathcal{S} \cup G$ and then it simply applies the default strategy for proving the formulas in G, described in Section 4.2.1. We have used `kcheck` to prove `gcd.cink` (Figure 4.5) as sketched in Example 4.2.1.

Except trivial examples, like `gcd` and `sum`, we tried our tool on more complex ones. Moreover, since our approach is parametric in language definitions, it can be applied to other $\mathbb{K}$ language definitions as well. Thus, in the following sections we describe our experience when verifying two non-trivial programs with `kcheck`, each experiment involving a different language and a different (type of) program. In Section 5.3.1 we have extended CinK with threads and we verified that a disjoint parallel program, called `FIND`, successfully finds (in parallel) the index of the first positive integer in an array. In Section 5.3.2, using the $\mathbb{K}$ definition of CinK and `kcheck` we verified the Knuth-Morris-Pratt [63] string matching algorithm.

5.3.1 Verifying a parallel program: `FIND`

The example is inspired from [2]. Given an integer array a and a constant $N \geq 1$, the program in Figure 5.5 finds the smallest index $k \in \{1, \ldots, N\}$

```
i = 1;                    S1 = while (i < oddtop){
j = 2;                           if (a[i] > 0) then{oddtop = i;}
oddtop = N + 1;                  else {i = i + 2;}
eventop = N + 1;                }
S1 || S2;                 S2 = while (j < eventop) {
if (oddtop > eventop)            if (a[j] > 0) then{eventop = j;}
  then { k = eventop; }          else{j = j + 2;}
  else { k = oddtop; }           }
```

Figure 5.5: `FIND` program.

such that $a[k] > 0$. If such an index k does not exists then $N + 1$ is returned. It is a *disjoint* parallel program, which means that its parallel components only have reading access to the variable a they share.

In order to verify `FIND`, we have enriched the $\mathbb{K}$ semantics of CinK with dynamic threads and the `||` operator, which executes in parallel two threads corresponding to `S1` and `S2`. Each thread $\langle\rangle_{\mathsf{th}}$ has its own computations $\langle\rangle_{\mathsf{k}}$ and environment $\langle\rangle_{\mathsf{env}}$ cells, while $\langle\rangle_{\mathsf{store}}$ is shared among the threads. Threads also have an $\langle\rangle_{\mathsf{id}}$ (identifier) cell. The configuration is shown below (the $^+$ on the $\langle\rangle_{\mathsf{th}^+}$ cell says that the cell contains at least one thread):

$$\langle \langle\!\langle \langle\cdot\rangle_{\mathsf{k}} \langle\cdot\rangle_{\mathsf{env}} \langle\cdot\rangle_{\mathsf{id}} \rangle\!\rangle_{\mathsf{th}^+} \langle\cdot\rangle_{\mathsf{store}} \langle\cdot\rangle_{\mathsf{result}} \langle\cdot\rangle_{\mathsf{stack}} \langle\cdot\rangle_{\mathsf{in}} \langle\cdot\rangle_{\mathsf{out}} \rangle_{\mathsf{cfg}}$$

The `||` operator yields a non-deterministic behaviour of `FIND`, i.e. it can generate more than one computation starting in a given initial state. However, in [2] the authors prove that all computations of a disjoint parallel program starting in the same initial state produce the same output. So, all the computations of a disjoint parallel program are equivalent to the sequential execution of its components. For program verification this observation simplifies matters because it allows independent verification of the parallel code, without considering all the interleavings caused by parallelism.

The verification of `FIND` is given by checking only three rules: one for each of the two loops and one for the main program. This is much simpler than the proof from [2], where more proof obligations must be generated and checked. This is a consequence of the fact that many proof obligations are automatically checked by symbolic execution. Moreover,

when performing mechanised verification, the pre/post conditions and the invariants must be very accurate. Otherwise, the proof will fail even if, intuitively, all the formulas seem to be valid. For example, when using `kcheck` to verify `FIND`, we discovered that the precondition *pre* must be $N \geq 1$ rather than *true* as stated in the (non-mechanised) proof of [2], and in p_2 the value of j must be greater-or-equal to 2, a constraint that was also forgotten in [2].

Figure 5.6 shows all the ingredients that we used to prove the correctness of the program `FIND` (Figure 5.5) using our tool. At the figure's top we show the code macros that we use in our RL formulas. Below the code macros we include the formulas corresponding to the pre/post conditions and invariants used by the authors of [2] in their proof. The program is checked by applying the implementation `kcheck` of our proof system on the consisting of the three RL formula-set $G = \{(\clubsuit), (\diamondsuit), (\spadesuit)\}$. On the bottom lines we show the proofs automatically constructed by `kcheck`.

We believe that the number of three proof obligations, given by G, is minimal for verifying `FIND`. Initially we started we eight rules describing the proof obligations used in [2]. Then, based on the `gcd` examples and others inspired from the same source, we realised that all sequential program fragment specifications can be removed since they can be automatically proved using the SymbolicStep rule, which, together with Transition and CaseAnalysis, amounts to symbolic execution. Since the configuration for the new language is more complex, the syntax for these rules is a bit cumbersome, but it can be generated from the annotations of the program by using symbolic execution to determine the exact structure of the configuration at the point where such a rule should be applied.

The proof trees for the RL formulas ($\clubsuit$) and ($\diamondsuit$) are similar to the one for the while-loop in the `gcd` program. However, here the second branch is split by a new use of the CaseAnalysis rule, due to the `if` statement from the loop's body. The proof tree for the RL formula ($\spadesuit$), corresponding to the specification of `FIND`, has a single branch because it uses circularities ($\clubsuit$) and ($\diamondsuit$) that do not split the proof tree.

The formulas are nontrivial, and it took us several iterations to come up with the exact ones, during which we used the tool in a trial-and-error process. The automatic nature of the tool, as well as the feedback it returned when it failed, were particularly helpful during this process. In

CODE MACROS

`INIT`	$\triangleq$	`i = 1; j = 2; oddtop = N + 1; eventop = N + 1;`
`BODY1`	$\triangleq$	`{if (a[i] > 0) then { oddtop = i; } else { i = i + 2; }}`
`BODY2`	$\triangleq$	`{if (a[j] > 0) then { eventop = j; } else { j = j + 2; }}`
`S1`	$\triangleq$	`while (i < oddtop) BODY1`
`S2`	$\triangleq$	`while (j < eventop) BODY2`
`MIN`	$\triangleq$	`if (oddtop > eventop) then { k = eventop; } else { k = oddtop; }`
`FIND`	$\triangleq$	`INIT S1||S2; MIN`

Formula macros

pre	$\triangleq$	$N \geq 1$
p_1	$\triangleq$	$1 \leq o \leq N+1 \wedge i\%2 = 1 \wedge 1 \leq i \leq o+1$
		$\wedge(\forall_{1 \leq l < i})(l\%2 = 1 \rightarrow a[l] \leq 0) \wedge (o \leq N \rightarrow a[o] > 0)$
p_1'	$\triangleq$	$1 \leq o' \leq N+1 \wedge i'\%2 = 1 \wedge 1 \leq i' \leq o'+1$
		$\wedge(\forall_{1 \leq l < i'})(l\%2 = 1 \rightarrow a[l] \leq 0) \wedge (o' \leq N \rightarrow a[o'] > 0)$
q_1	$\triangleq$	$1 \leq o' \leq N+1 \wedge (\forall_{1 \leq l < o'})(l\%2 = 1 \rightarrow a[l] \leq 0) \wedge (o' \leq N \rightarrow a[o'] > 0)$
p_2	$\triangleq$	$2 \leq e \leq N+1 \wedge j\%2 = 0 \wedge 2 \leq j \leq e+1$
		$\wedge(\forall_{1 \leq l < j})(l\%2 = 0 \rightarrow a[l] \leq 0) \wedge (e \leq N \rightarrow a[e] > 0)$
p_2'	$\triangleq$	$2 \leq e' \leq N+1 \wedge j'\%2 = 0 \wedge 2 \leq j' \leq e'+1$
		$\wedge(\forall_{1 \leq l < j'})(l\%2 = 0 \rightarrow a[l] \leq 0) \wedge (e' \leq N \rightarrow a[e'] > 0)$
q_2	$\triangleq$	$2 \leq e' \leq N+1 \wedge (\forall_{1 \leq l < e'})(l\%2 = 0 \rightarrow a[l] \leq 0) \wedge (e' \leq N \rightarrow a[e'] > 0)$
$post$	$\triangleq$	$1 \leq k' \leq N+1 \wedge (\forall_{1 \leq l < k'})(a[l] \leq 0) \wedge (k' \leq N \rightarrow a[k'] > 0)$

Map macros for environment and store

Env	$\triangleq$	$\mathtt{a} \mapsto \mathbf{a}\ \mathtt{i} \mapsto \mathbf{i}\ \mathtt{j} \mapsto \mathbf{j}\ \mathtt{oddtop} \mapsto \mathbf{o}\ \mathtt{eventop} \mapsto \mathbf{e}\ \mathtt{N} \mapsto \mathbf{N}\ \mathtt{k} \mapsto \mathbf{k}$
St	$\triangleq$	$\mathbf{a} \mapsto a\ \mathbf{i} \mapsto i\ \mathbf{j} \mapsto j\ \mathbf{o} \mapsto o\ \mathbf{e} \mapsto e\ \mathbf{N} \mapsto N\ \mathbf{k} \mapsto k$
St'	$\triangleq$	$\mathbf{a} \mapsto a\ \mathbf{i} \mapsto i'\ \mathbf{j} \mapsto j'\ \mathbf{o} \mapsto o'\ \mathbf{e} \mapsto e'\ \mathbf{N} \mapsto N\ \mathbf{k} \mapsto k'$

RL formulas

(♣)	$\langle\langle \mathtt{S1} \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St \rangle_{\mathsf{st}} \wedge i < o \wedge p_1$	$\Rightarrow$	$\langle\langle \cdot \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St' \rangle_{\mathsf{st}} \wedge o' \leq i' \wedge p_1' \wedge q_1$
(♢)	$\langle\langle \mathtt{S2} \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St \rangle_{\mathsf{st}} \wedge j < e \wedge p_2$	$\Rightarrow$	$\langle\langle \cdot \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St' \rangle_{\mathsf{st}} \wedge e' \leq j' \wedge p_2' \wedge q_2$
(♠)	$\langle\langle \mathtt{FIND} \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St \rangle_{\mathsf{st}} \wedge pre$	$\Rightarrow$	$\langle\langle \cdot \rangle_{\mathsf{k}} \langle Env \rangle_{\mathsf{env}} \rangle_{\mathsf{th}} \langle St' \rangle_{\mathsf{st}} \wedge post$

Corresponding proofs given by `kcheck`

$s(i) \triangleq$ [CaseAnalysis], ([SymbolicStep]) $\vee$ ([SymbolicStep]),[CircularHypothesis](i)
(♣) [SymbolicStep], [CaseAnalysis], [Implication] $\vee$ (s(♣), [Implication])
(♢) [SymbolicStep], [CaseAnalysis], [Implication] $\vee$ (s(♢), [Implication])
(♠) [SymbolicStep] $\times 5$, [CircularHypothesis](1), [CircularHypothesis](2), [Implication]

Figure 5.6: RL formulas necessary to verify `FIND`. We use `a`, `i`, `j`, `oddtop`, `eventop`, `N`, `k` to denote program variables, **a**, **i**, **j**, **o**, **e**, **N**, **k** to denote locations, and a, i, j, o, e, N, k for variables values. We also use $s(i)$ to denote a common sequence in the proofs of (♣) and (♢). CaseAnalysis splits the proof in two goals separated by $\vee$, while CircularHypothesis(i) represents the application of the formula (i) as a circularity. [SymbolicStep] $\times n$ is the equivalent of applying [SymbolicStep] n times.

particular symbolic execution was fruitfully used for the initial testing of programs before they were verified.

5.3.2 Verifying the Knuth-Morris-Pratt string matching algorithm: KMP

The Knuth-Morris-Pratt algorithm [63] searches for occurrences of a word P, usually called *pattern*, within a given text T by making use of the fact that when a mismatch occurs, the pattern contains sufficient information to determine where the next search should begin. A detailed description of the algorithm, whose CinK code is shown in Figure 5.7, can be found in [30].

The KMP algorithm optimises the naive search of a pattern into a given string by using some additional information collected from the pattern. For instance, let us consider $T =$ ABADABCDA and $P =$ ABAC. It can be easily observed that ABAC does not match ABADABCDA starting with the first position because there is a mismatch on the fourth position, namely $\texttt{C} \neq \texttt{D}$.

The KMP algorithm uses a *failure function* π, which, for each position j in P, returns the length of the longest proper prefix of the pattern which is also a suffix of it. For our example, $\pi[3] = 1$ and $\pi[j] = 0$ for $j = 1, 2, 4$. In the case of a mismatch between the position i in T and the position j in P, the algorithm proceeds with the comparison of the positions i and $\pi[j]$. For the above mismatch, the next comparison is between the B in ABAC and the first instance of D in ABADABCDA, which saves a comparison of the characters preceeding them, since the algorithm "already knows" that they are equal (here, they are both A).

An implementation of KMP is shown in Figure 5.7. The comments include the specifications for preconditions, postconditions, and invariants, which will be explained later in this section (briefly, they are syntactic sugar for RL formulas, which are automatically generated from them). The program can be run either using the $\mathbb{K}$ semantics of CinK or the g++ GNU compiler. The compute_prefix function computes the failure function π for each component of the pattern and stores it in a table, called pi. The kmp_matcher searches for all occurrences of the pattern in the string comparing characters one by one; when a mismatch is found on positions i in the string and q in the pattern, the algorithm shifts the search to the right as many positions as indicated by pi[q], and initiates a

new search. The algorithm stops when the string is completely traversed.

For the proof of KMP we use the original algorithm as presented in [30]. Another formal proof of the algorithm is given in [40] by using Why3 [41]. There, the authors collapsed the nested loops into a single one in order to reduce the number of invariants they have to provide. They also modified the algorithm to stop when the first occurrence of the pattern in the string was found. By contrast, we do not modify the algorithm from [30]. We also prove that KMP finds *all* the occurrences of the pattern in the string, not only the first one. We let $P[1..i]$ denote the prefix of P of size i, and $P[i]$ denote its i-th element.

Definition 5.3.1 *Let P be a pattern of size $m \geq 1$ and T a string of characters of size $n \geq 1$. We define the following functions and predicate:*

- *$\pi(i)$ is the length of the longest proper prefix of $P[1..i]$ which is also a suffix for $P[1..i]$, for all $1 \leq i \leq m$;*
- *$\theta(i)$ is the length of the longest prefix of P that matches T on the final position i, for all $1 \leq i \leq n$;*
- *$allOcc(Out, P, T, i)$ holds iff the list Out contains all the occurrences of P in $T[1..i]$.*

The specification of the `kmp_matcher` function is the following RL formula:

$$\left\langle \begin{array}{l} \langle \texttt{kmp\_matcher}(\texttt{p},\texttt{t},m,n);\rangle_{\mathsf{k}} \langle\cdot\rangle_{\mathsf{out}} \\ \langle \texttt{p} \mapsto l_1 \ \texttt{t} \mapsto l_2 \rangle_{\mathsf{env}} \langle l_1 \mapsto P \ l_2 \mapsto T \rangle_{\mathsf{store}} \end{array} \ \ldots \right\rangle_{\mathsf{cfg}} \wedge\, n \geq 1 \wedge m \geq 1$$

$$\Rightarrow$$

$$\langle \langle\cdot\rangle_{\mathsf{k}} \langle Out \rangle_{\mathsf{out}} \langle \ldots \rangle_{\mathsf{env}} \langle \ldots \rangle_{\mathsf{store}} \ldots \rangle_{\mathsf{cfg}} \wedge \quad allOcc(Out, P, T, n)$$

This formula says that from a configuration where the program variables `p` and `t` are bound to the values P, T, respectively, the output cell is empty, and the `kmp_matcher` function has to be executed, one reaches a configuration where the function has been executed and the output cell contains all the occurrences of P in T. Note that we passed the symbolic values m and n as actual parameters to the function which are the sizes of P, and T, respectively. An advantage of RL with respect to Hoare

```
/*@pre: m>=1 */
void compute_prefix(char p[],
                    int m, int pi[])
{
  int k, q;
  k = 0;
  pi[1] = 0;
  q = 2;
  while(q <= m) {
  /*@inv: 0<=k /\ k<q /\ q<=m+1 /\
    (forall u:1..k)(p[u]=p[q-k+u]) /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    Pi(q)<=k+1 */
    while (k > 0 && p[k+1] != p[q]) {
    /*@inv: 0<=k /\ k<q /\ q<=m /\
      (forall u:1..k)(p[u]=p[q-k+u]) /\
      (forall u:1..q-1)(pi[u]=Pi(u)) /\
      (forall u:1..m)(0<=Pi(u)<u) /\
      Pi(q)<=k+1  */
      k = pi[k];
    }
    if (p[k + 1] == p[q]) {
      k = k + 1;
    }
    pi[q] = k;
    q++;
  }
}
/*@post: (forall u:1..m)(pi[u]=Pi(u)) */
```

```
/*@pre: m>=1 /\ n>=1 */
void kmp_matcher(char p[], char t[], int m, int n)
{
  int q = 0, i = 1, pi[m];
  compute_prefix(p, m, pi);
  while (i <= n) {
  /*@inv: 1<=m /\ 0<=q<=m /\ 1<=i<=n+1 /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    (exists v)(forall u:v+1..i-1)(Theta(u)<m /\
                          allOcc(Out,p,t,v))/\
    (forall u:1..q)(p[u]=t[i-1-q+u]) /\
    Theta(i)<=q+1 */
    while (q > 0 && p[q + 1] != t[i]) {
    /*@inv: 1<=m /\ 0<=q /\ q<m /\
    (forall u:1..q-1)(pi[u]=Pi(u)) /\
    (exists v)(forall u:v+1..i-1)(Theta(u)<m /\
                          allOcc(Out,p,t,v))/\
    (forall u:1..q)(p[u]=t[i-1-q+u]) /\
    (forall u:1..i-1)(Theta(u)<m) /\
    Theta(i)<=q+1 */
      q = pi[q];
    }
    if (p[q + 1] == t[i]) { q = q + 1;  }
    if (q == m) {
      cout << "shift: " << (i - m) << endl;
      q = pi[q];
    }
    i++;
  }}
/*@post: allOcc(Out, p, t, n) */
```

Figure 5.7: The KMP algorithm annotated with pre-/post-conditions and invariants: failure function (left) and the main function (right). Note that we used `Pi`, `Theta`, and `allOcc` to denote functions π and θ, and predicate *allOcc*, respectively.

Logic is, in addition to language independence, the fact that RL formulas may refer to all the language's configuration, whereas Hoare Logic formulas may only refer to program variables. A Hoare Logic formula for the `kmp_matcher` function would require the addition of assignments to a new variable playing the role of our output cell.

There are some additional issues concerning the way users write the RL formulas. These may be quite large depending on the size of the $\mathbb{K}$ configuration of the language. To handle that, we have created an interactive tool for generating such formulas. Users can annotate their programs with preconditions and postconditions and then use our tool to generate RL formulas from those annotations. The above specification for `KMP` is generated from the annotations:

```
//@pre: m >= 1 /\ n >= 1
kmp_matcher(p, t, m, n);
//@post: allOcc(Out, p, t, n)
```

Loops can be annotated with invariants as shown below:

```
while (COND) {
  //@inv: INV
  k = pi[k];
}
```

For each annotated loop, the tool generates a RL formula which states that by starting with a configuration where the entire loop remains to be executed and `INV` holds, one reaches a configuration where the loop was completely executed and $\texttt{INV} \wedge \neg\texttt{COND}$ holds.

From the annotations shown in Figure 5.7 the tool generates all the RL formulas that we need to prove `KMP`. Since `KMP` has four loops and two pairs of pre/post-conditions, the tool generates and proves a total number of six RL formulas. In the annotations we use the program variables (e.g. `pi`, `p`, `m`) and a special variable `Out` which is meant to refer the content of the $\langle\rangle_{\mathsf{out}}$ cell. This variable gives us access to the output cell, which is essential in proving that the algorithm computes all the occurrences of the pattern.

Finally, since every particular verification problem requires problem-specific constructions and properties about them, for verifying `KMP` we have enriched the symbolic definition of CinK with functional symbols for π, θ, and *allOcc*, and some of their properties shown in Figure 5.8.

1. $0 \leq k \leq m \vdash 0 \leq \pi(k) < k$.

2. $0 \leq q \leq n \vdash 0 \leq \theta(q) \leq m$.

3. $(\forall u : 1..k)(P[u] = P[q-k+u]) \wedge \pi(q) \leq k+1 \wedge P[k+1] \neq P[q] \vdash$
$\pi(q) \leq \pi(k)+1$.

4. $(\forall u : 1..k)(P[u] = P[q-k+u]) \wedge \pi(q) \leq k+1 \wedge P[k+1] = P[q] \vdash$
$\pi(i) = k+1$.

5. $(\forall u : 1..q)(P[u] = T[i-1-q+u]) \wedge \theta(i) \leq q+1 \wedge P[q+1] \neq T[i] \vdash$
$\theta(i) \leq \pi(q)+1$.

6. $(\forall u : 1..q)(P[u] = T[i-1-q+u]) \wedge \theta(i) \leq q+1 \wedge P[q+1] = T[i] \vdash$
$\theta(i) = q+1$.

7. $(\exists v)(\forall u : v{+}1..i{-}1)(allOcc(Out, P, T, v) \wedge \theta(u) < m) \wedge \theta(i) = m \wedge i < n \vdash$
$(\exists v)(\forall u : v{+}1..i)(allOcc(Out, P, T, v) \wedge \theta(u) < m)$.

8. $(\exists v)(\forall u : v{+}1..i)(allOcc(Out, P, T, v) \wedge \theta(u) < m) \wedge i = n \vdash$
$allOcc(Out, P, T, v)$.

Figure 5.8: Properties needed to prove `KMP`.

Chapter 6

Conclusions

We have presented a formal and generic framework for the symbolic execution of programs in languages definable in an algebraic and term-rewriting setting. Symbolic execution is performed by applying the rules in the semantics of a language by so-called symbolic rewriting. We prove that the symbolic execution thus defined has the naturally expected properties with respect to concrete execution: *coverage*, meaning that to each concrete execution there is a feasible symbolic on on the same program path, and *precision*, meaning that each feasible symbolic execution has a concrete execution on the program same path. These properties are expressed in terms of mutual simulations.

In order to implement our symbolic rewriting-based approach in a setting where only standard execution is available, such as the $\mathbb{K}$ framework, we define a transformation of language definitions $\mathcal{L}$ into other language definitions $\mathcal{L}^{\mathfrak{s}}$, and show that concrete program execution $\mathcal{L}^{\mathfrak{s}}$, which uses standard rewriting, are in a covering&precise relationship with the symbolic execution of the corresponding programs in $\mathcal{L}$.

The incorporation of symbolic execution into a deductive system for program verification with respect to Reachability-Logic specifications is also presented in detail. Reachability Logic has a sound and relatively complete deduction system, which offers a lot of freedom (but very few guidelines) for constructing proofs. We came up with an alternative proof system, and we show that, under reasonable conditions on the semantics of programming languages and of the Reachability Logic formulas, a cer-

tain strategy executing our proof system is sound and weakly complete. This essentially means that, when it terminates, the strategy solves the Reachability-Logic verification problem: when presented with a valid input (set of RL formulas) it proves the formulas, and when presented with an invalid input it detects this invalidity.

Finally, we present the implementation of a prototype tool for symbolic execution based on the above theory, which is now a part of the $\mathbb{K}$ framework. We illustrate the tool's capabilities on several programs written in different programming languages, in order to emphasise that our framework is not language dependent. Moreover, we show that the tool can be easily extended to perform other types of analyses, e.g. bounded model checking, LTL model-checking, etc. We also introduce a prototype for Reachability-Logic verification based on our proof system, which is implemented on top of our symbolic execution. We used this prototype by proving several non-trivial programs.

The work presented in this dissertation is important for at least three reasons: first, we formalise and prove two important properties of symbolic execution which allows one to consistently transfer properties of symbolic execution to concrete executions; second, we have shown that this framework is suitable for language independent program verification; finally, we provide an implementation which shows that language independent symbolic execution based on formal semantics is possible.

6.1 Future Work

We are planning to expand our tool, to make it able to seamlessly perform a wide range of program analyses, from testing and debugging to formal verifications, following ideas presented in related work, but with the added value of being language independent and grounded in formal methods. For this, we shall develop a rich domain of symbolic values, able to handle various kinds of data types. Formalising the interaction of symbolic-domain computations with symbolic execution is also a matter for future work.

Another future research direction is specifically targeted at our RL-formulas verifier, and aims at certifying its executions. The idea is to generate proof scripts for the Coq proof assistant [73], in order to obtain

certificates that, despite any (inevitable) bugs in our tool, the proofs it generates are indeed correct. This amounts to, firstly, encoding our RL proof system in Coq, and proving its soundness with respect to the original proof system of RL (which have already been proved sound in Coq [93]). Secondly, our verifier must be enhanced to return, for any successful execution, the rules of our system it has applied and the substitutions it has used. From this information a Coq script is built that, if successfully run by Coq, generates a proof term that constitutes a correctness certificate for the verifier's original execution. A longer-term objective is to turn our verifier into an external proof tactic for Coq, resulting in a powerful mixed interactive/automatic program verification tool.

Bibliography

[1] W. Ahrendt. The KeY tool. *Software and Systems Modeling*, 4:32–54, 2005.

[2] Krzysztof R. Apt, Frank de Boer, and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer Verlag, 3rd edition, 2009.

[3] Alessandro Armando, Massimo Benerecetti, and Jacopo Mantovani. Model checking linear programs with arrays. In *Proceedings of the Workshop on Software Model Checking*, volume 144 - 3, pages 79 – 94, 2006.

[4] Andrei Arusoaie. Engineering hoare logic-based program verification in K framework. In *SYNASC*, pages 177–184, 2013.

[5] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. A generic framework for symbolic execution. In *6th International Conference on Software Language Engineering*, volume 8225 of *LNCS*, pages 281–301. Springer Verlag, 2013. Also available as a technical report `http://hal.inria.fr/hal-00853588`.

[6] Andrei Arusoaie, Dorel Lucanu, and Vlad Rusu. Towards a semantics for OCL. *Electronic Notes in Theoretical Computer Science*, 304(0):81 – 96, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[7] Andrei Arusoaie, Traian Florin Şerbănuţă, Chucky Ellison, and Grigore Roşu. Making Maude definitions more interactive. In

Rewriting Logic and Its Applications, WRLA 2012, volume 7571 of *Lecture Notes in Computer Science*. Springer, 2012.

[8] Irina Mariuca Asavoae. *K Semantics for Abstractions*. PhD thesis, Alexandru Ioan Cuza, Univerisity of Iaşi, September 2012. PhD Thesis.

[9] Irina Mariuca Asavoae and Mihail Asavoae. Collecting semantics under predicate abstraction in the K framework. In *WRLA*, pages 123–139, 2010.

[10] Irina Mariuca Asavoae, Mihail Asavoae, and Dorel Lucanu. Path directed symbolic execution in the K framework. In *SYNASC*, pages 133–141, 2010.

[11] Mihail Asavoae. *Semantics-Based WCET Analysis*. PhD thesis, Alexandru Ioan Cuza, Univerisity of Iaşi, September 2012. PhD Thesis.

[12] Mihail Asavoae, Irina Mariuca Asavoae, and Dorel Lucanu. On abstractions for timing analysis in the K framework. In *Proceedings of the Second International Conference on Foundational and Practical Aspects of Resource Analysis*, FOPARA'11, pages 90–107, Berlin, Heidelberg, 2012. Springer-Verlag.

[13] Irina Măriuca Asăvoae. Abstract semantics for alias analysis in k. *Electronic Notes in Theoretical Computer Science*, 304(0):97 – 110, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[14] Mihail Asăvoae. K semantics for assembly languages: A case study. *Electronic Notes in Theoretical Computer Science*, 304(0):111 – 125, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[15] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: an overview. In *Proc. 2004 international conference on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, CASSIS'04, pages 49–69, 2005.

[16] Patrick Baudin, Jean C. Filliâtre, Thierry Hubert, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI C Specification Language (preliminary design V1.2)*, preliminary edition, May 2008.

[17] Denis Bogdănaş. Java semantics in $\mathbb{K}$. `https://github.com/kframework/java-semantics`.

[18] Sylvie Boldo and Jean-Christophe Filliâtre. Formal verification of floating-point programs. In *IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE Computer Society, 2007.

[19] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. SELECT-a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software*, pages 234–245, New York, NY, USA, 1975. ACM.

[20] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 239–254, New York, NY, USA, 2014. ACM.

[21] Stefan Bucur, Johannes Kinder, and George Candea. Prototyping symbolic execution engines for interpreted languages. *SIGPLAN Not.*, 49(4):239–254, February 2014.

[22] Jacob Burnim and Koushik Sen. Heuristics for scalable dynamic test generation. In *ASE*, pages 443–446. IEEE, 2008.

[23] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[24] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), 2008.

[25] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. S2E: A platform for in-vivo multi-path analysis of software systems. *SIGPLAN Not.*, 47(4):265–278, March 2011.

[26] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The S2E platform: Design, implementation, and applications. *ACM Trans. Comput. Syst.*, 30(1):2:1–2:49, February 2012.

[27] Edmund Clarke and Daniel Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ASP-DAC '03, pages 308–311, New York, NY, USA, 2003. ACM.

[28] Lori A. Clarke. A program testing system. In *Proceedings of the 1976 Annual Conference*, ACM '76, pages 488–491, New York, NY, USA, 1976. ACM.

[29] Alberto Coen-Porisini, Giovanni Denaro, Carlo Ghezzi, and Mauro Pezzé. Using symbolic execution for verifying safety-critical systems. *SIGSOFT Softw. Eng. Notes*, 26(5):142–151, 2001.

[30] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[31] Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. Verifying systems rules using rule-directed symbolic execution. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 329–342, New York, NY, USA, 2013. ACM.

[32] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[33] Jakub Daniel, Pavel Parizek, and Corina S. Pasareanu. Predicate abstraction in Java Pathfinder. *ACM SIGSOFT Software Engineering Notes*, 39(1):1–5, 2014.

[34] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[35] Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.

[36] Chucky Ellison, Traian Florin Şerbănuţă, and Grigore Roşu. A rewriting logic approach to type inference. In *Recent Trends in Algebraic Development Techniques*, volume 5486 of *Lecture Notes in Computer Science*, pages 135–151. Springer, 2009. Revised Selected Papers from the 19th International Workshop on Algebraic Development Techniques (WADT'08).

[37] Santiago Escobar, José Meseguer, and Ralf Sasse. Variant narrowing and equational unification. *Electr. Notes Theor. Comput. Sci.*, 238(3):103–119, 2009.

[38] PEX: Automated exploratory testing for .NET.

[39] Daniele Filaretti and Sergio Maffeis. An executable formal semantics of PHP. In *Proceedings of European Conference on Object-Oriented Programming*, 2014.

[40] Jean Christophe Filliâtre. Proof of KMP string searching algorithm. `http://toccata.lri.fr/gallery/kmp.en.html`.

[41] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[42] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages

19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[43] Werner Gabrisch and Wolf Zimmermann. A Hoare-style verification calculus for control state ASMs. In *Proceedings of the Fifth Balkan Conference in Informatics*, BCI '12, pages 205–210, New York, NY, USA, 2012. ACM.

[44] Milos Gligoric, Darko Marinov, and Sam Kamin. Codese: Fast deserialization via code generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 298–308, New York, NY, USA, 2011. ACM.

[45] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA, 2005. ACM.

[46] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, 2012.

[47] Joseph A. Goguen and José Meseguer. Order-sorted algebra I: Equational deduction for multiple inheritance, overloading, exceptions and partial operations. *Theor. Comput. Sci.*, 105(2):217–273, 1992.

[48] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[49] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, History of Computing, chapter 5, pages 101–121. Springer London, London, 2010.

[50] K. Goshi, P. Wray, and Yong Sun. An intelligent tutoring system for teaching and learning Hoare logic. In *Computational Intelligence and Multimedia Applications, 2001. ICCIMA 2001. Proceedings. Fourth International Conference on*, pages 293–297, 2001.

[51] Dwight Guth. A formal semantics of Python 3.3. Master's thesis, University of Illinois at Urbana-Champaign, July 2013.

[52] David Harel, Dexter Kozen, and Jerzy Tiuryn. Dynamic logic. In *Handbook of Philosophical Logic*, pages 497–604. MIT Press, 1984.

[53] Mark Hills and Grigore Roşu. KOOL: An Application of Rewriting Logic to Language Prototyping and Analysis. In *Proceedings of the 18th International Conference on Rewriting Techniques and Applications (RTA'07)*, volume 4533 of *LNCS*, pages 246–256. Springer, 2007.

[54] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[55] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[56] W.E. Howden. Symbolic testing and the dissect symbolic evaluation system. *Software Engineering, IEEE Transactions on*, SE-3(4):266–278, July 1977.

[57] ISO-IEC. Standard for Programming Language C++. Working Draft. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3797.pdf.

[58] J-Algo. The algorithm visualization tool, 2007.

[59] Bart Jacobs, Jan Smans, and Frank Piessens. A quick tour of the verifast program verifier. In *Proceedings of the 8th Asian conference on Programming languages and systems*, APLAS'10, pages 304–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[60] Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. Tracer: a symbolic execution tool for verification. In *Proc. 24th international conference on Computer Aided Verification*, CAV'12, pages 758–766. Springer-Verlag, 2012.

[61] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In Hubert

Garavel and John Hatcliff, editors, *TACAS*, volume 2619 of *Lecture Notes in Computer Science*, pages 553–568. Springer, 2003.

[62] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[63] Donald E. Knuth, J.H. Morris, and Vaughan R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, 1977.

[64] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *SIGPLAN Not.*, 47(6):193–204, June 2012.

[65] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[66] David Lazar. K definition of Haskell, 2013.

[67] David Lazar, Andrei Arusoaie, Traian Florin Serbanuta, Chucky Ellison, Radu Mereuta, Dorel Lucanu, and Grigore Roşu. Executing formal semantics with the K tool. In *FM*, volume 7436 of *LNCS*, pages 267–271. Springer, 2012.

[68] David Lazar and Chucky Ellison. K definition of the llvm assembly language, 2012.

[69] Dorel Lucanu and Vlad Rusu. Program equivalence by circular reasoning. In *IFM*, Lecture Notes in Computer Science, pages 362–377. Springer, 2013.

[70] Dorel Lucanu and Traian Florin Serbanuta. CinK - an exercise on how to think in K. Technical Report TR 12-03, Version 2, Alexandru Ioan Cuza University, Faculty of Computer Science, December 2013.

[71] Dorel Lucanu, Traian Florin Şerbănuţă, and Grigore Roşu. The K Framework distilled. In *9th International Workshop on Rewriting Logic and its Applications*, volume 7571 of *Lecture Notes in Computer Science*, pages 31–53. Springer, 2012. Invited talk.

[72] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. Directed symbolic execution. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 95–111, Berlin, Heidelberg, 2011. Springer-Verlag.

[73] Reference Manual. The Coq proof assistant. `http://coq.inria.fr/refman/`.

[74] The $\mathbb{K}$ tool. Github repository. `https://github.com/kframework/k`.

[75] Daniel D. McCracken and Edwin D. Reilly. Backus-naur form (bnf). In *Encyclopedia of Computer Science*, pages 129–131. John Wiley and Sons Ltd., Chichester, UK, 2003.

[76] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010.

[77] José Meseguer. Rewriting logic and Maude: Concepts and applications. In Leo Bachmair, editor, *RTA*, volume 1833 of *LNCS*, pages 1–26. Springer, 2000.

[78] José Meseguer and Prasanna Thati. Symbolic reachability analysis using narrowing and its application to verification of cryptographic protocols. *Higher-Order and Symbolic Computation*, 20(1-2):123–160, 2007.

[79] José Meseguer. Rewriting as a unified model of concurrency. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90 Theories of Concurrency: Unification and Extension*, volume 458 of *Lecture Notes in Computer Science*, pages 384–400. Springer Berlin Heidelberg, 1990.

[80] Daejun Park. K definition of Javascript, 2013.

[81] Corina S. Pasareanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael R. Lowry, Suzette Person, and Mark Pape. Combining unit-level symbolic execution and system-level concrete

execution for testing nasa software. In Barbara G. Ryder and Andreas Zeller, editors, *ISSTA*, pages 15–26. ACM, 2008.

[82] Corina S. Păsăreanu and Neha Rungta. Symbolic pathfinder: symbolic execution of Java bytecode. In *ASE*, pages 179–180, 2010.

[83] Corina S. Pasareanu and Willem Visser. Verification of java programs using symbolic execution and invariant generation. In Susanne Graf and Laurent Mounier, editors, *SPIN*, volume 2989 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2004.

[84] Corina S. Pasareanu, Willem Visser, David H. Bushnell, Jaco Geldenhuys, Peter C. Mehlitz, and Neha Rungta. Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis. *Autom. Softw. Eng.*, 20(3):391–425, 2013.

[85] Grigore Roşu Patrick Meredith, Mark Hills. An Executable Rewriting Logic Semantics of K-Scheme. In Danny Dube, editor, *Proceedings of the 2007 Workshop on Scheme and Functional Programming (SCHEME'07), Technical Report DIUL-RT-0701*, pages 91–103. Laval University, 2007.

[86] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. Directed incremental symbolic execution. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 504–515, New York, NY, USA, 2011. ACM.

[87] David A Ramos and Dawson R. Engler. Practical, low-effort equivalence verification of real code. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 669–685, Berlin, Heidelberg, 2011. Springer-Verlag.

[88] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[89] Camilo Rocha, José Meseguer, and Cesar A. Munoz. Rewriting modulo SMT. Technical Report 218033, NASA, 2013.

[90] Camilo Rocha, José Meseguer, and Cesar A. Munoz. Rewriting modulo smt and open system analysis. In *Proceedings of the 10th International Workshop on Rewriting Techniques and Applications*, WRLA'14. LNCS, 2014.

[91] Grigore Roşu. CS322, Fall 2003 - Programming Language Design: Lecture Notes. Technical Report UIUCDCS-R-2003-2897, University of Illinois at Urbana-Champaign, Department of Computer Science, December 2003. Lecture notes of a course taught at UIUC.

[92] Grigore Roşu and Andrei Ştefănescu. Matching logic rewriting: Unifying operational and axiomatic semantics in a practical and generic framework. Technical Report http://hdl.handle.net/2142/28357, University of Illinois, November 2011.

[93] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.

[94] Grigore Roşu, Chucky Ellison, and Wolfram Schulte. Matching logic: An alternative to Hoare/Floyd logic. In Michael Johnson and Dusko Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology And Software Technology (AMAST '10)*, volume 6486 of *Lecture Notes in Computer Science*, pages 142–162, 2010.

[95] Grigore Roşu and Dorel Lucanu. Circular coinduction – a proof theoretical foundation. In *CALCO 2009*, volume 5728 of *LNCS*, pages 127–144. Springer, 2009.

[96] Grigore Rosu, Wolfram Schulte, and Traian Florin Serbanuta. Runtime verification of C memory safety. In Saddek Bensalem and Doron A. Peled, editors, *Runtime Verification (RV'09)*, volume 5779 of *Lecture Notes in Computer Science*, pages 132–152, 2009.

[97] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[98] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 555–574. ACM, 2012.

[99] Grigore Roşu and Andrei Ştefănescu. Checking reachability using matching logic. Technical Report http://hdl.handle.net/2142/33771, UIUC, August 2012.

[100] Grigore Rosu and Andrei Stefanescu. From hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.

[101] Grigore Rosu and Andrei Stefanescu. Towards a unified theory of operational and axiomatic semantics. In *Proceedings of the 39th International Colloquium on Automata, Languages and Programming (ICALP'12)*, volume 7392 of *Lecture Notes in Computer Science*, pages 351–363. Springer, 2012.

[102] Grigore Roşu and Traian Florin Şerbănuţă. Overview and SIMPLE case study. *Electronic Notes in Theoretical Computer Science*, 304(0):3 – 56, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[103] Samir Sapra, Marius Minea, Sagar Chaki, Arie Gurfinkel, and Edmund M. Clarke. Finding errors in python programs using dynamic symbolic execution. In *ICTSS*, pages 283–289, 2013.

[104] Ralf Sasse and José Meseguer. Java+ITP: A verification tool based on Hoare Logic and Algebraic Semantics. *Electron. Notes Theor. Comput. Sci.*, 176(4):29–46, July 2007.

[105] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Chicago, IL, July 2009.

[106] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and

Robert B. Jones, editors, *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[107] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[108] Traian Florin Şerbănuţă. *A Rewriting Approach to Concurrent Programming Language Design and Semantics*. PhD thesis, University of Illinois at Urbana-Champaign, December 2010. `https://www.ideals.illinois.edu/handle/2142/18252`.

[109] Traian Florin Serbanuta, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Rosu. The k primer (version 3.3). In *Proceedings of International K Workshop (K'11)*, ENTCS. Elsevier, 2013. To appear.

[110] Traian Florin Serbanuta and Grigore Rosu. A truly concurrent semantics for the k framework based on graph transformations. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *ICGT*, volume 7562 of *Lecture Notes in Computer Science*, pages 294–310. Springer, 2012.

[111] Traian Florin Şerbănuţă, Grigore Roşu, and José Meseguer. A rewriting logic approach to operational semantics. *Information and Computation*, 207:305–340, 2009.

[112] Cristian Zamfir and George Candea. Execution synthesis: A technique for automated software debugging. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 321–334, New York, NY, USA, 2010. ACM.

[113] Traian Florin Şerbănuţă. Rewriting semantics and analysis of concurrency features for a C-like language. *Electronic Notes in Theoretical Computer Science*, 304(0):167 – 182, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).

[114] Traian Florin Şerbănuţă, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The primer (version 3.3). *Electronic Notes in Theoretical Computer Science*, 304(0):57 – 80, 2014. Proceedings of the Second International Workshop on the K Framework and its Applications (K 2011).